Adobe

Adobe FrameMaker

MIF Reference Guide

March 2026

Contents

Chapter 1: Introduction

Why use MIF? 1
Using this manual 1
Style conventions 2
Overview of MIF statements 2
MIF statement syntax 4

Chapter 2: Using MIF Statements

Working with MIF files 9
Creating a simple MIF file for FrameMaker 11
Creating and applying character formats 24
Creating and formatting tables 25
Specifying page layout 32
Creating markers 37
Creating cross-references 37
Creating variables 39
Creating conditional text 41
Creating filters 44
Including template files 45
Setting View Only document options 47
Applications of MIF 48
Debugging MIF files 51
Other application tools 52
Where to go from here 52

Chapter 3: MIF Document Statements

MIF file layout 53
MIFFile statement 55
Macro statements 56
Track edited text 57
Conditional text 57
Boolean expressions 59
Filter By Attribute 60
Paragraph formats 61
Character formats 66
Object styles 71
Line numbers 74
Tables 74
Color 84
Variables 87
Cross-references 87
Global document properties 88

Pages 109
Mini TOC 110
Graphic objects and graphic frames 111
Text flows 130
Text insets (text imported by reference) 138

Chapter 4: MIF Book File Statements
MIF book file overview 145
MIF book file identification line 146
Book statements 146

Chapter 5: MIF Statements for Structured Documents and Books
Structural element definitions 157
Attribute definitions 160
Format rules 162
Format change lists 168
Elements 174
Banner text 177
Filter By Attribute 177
XML data for structured documents 178
Preference settings for structured documents 179
Text in structured documents 182
Structured book statements 182
MIF Messages 186

Chapter 6: MIF Equation Statements
MathML statement 188
Document statement 189
Math statement 193
MathFullForm statement 194

Chapter 7: MIF Asian Text Processing Statements
Asian Character Encoding 213
Combined Fonts 214
Kumihan Tables 217
Rubi text 227

Chapter 8: Examples
Text example 231
Bar chart example 232
Pie chart example 235
Custom dashed lines 236
Table examples 238
Database publishing 241

Chapter 9: MIF Messages
General form for MIF messages 248

List of MIF messages 248

Chapter 10: MIF Compatibility

Changes between version 12.0 and 2015 release 251
Changes between version 11.0 and 12.0 253
Changes between version 9.0 and 10.0 254
MIF syntax changes in FrameMaker 8 254
Changes between version 6.0 and 7.0 255
Changes between version 5.5 and 6.0 256
Changes between version 5 and 5.5 257
Changes between versions 4 and 5 259
Changes between versions 3 and 4 262

Chapter 11: Facet Formats for Graphics

Facets for imported graphics 267
Basic facet format 268
Graphic insets (UNIX versions) 269
General rules for reading and writing facets 274

Chapter 12: EPSI Facet Format

Specification of an EPSI facet 275
Example of an EPSI facet 275

Chapter 13: FrameImage Facet Format

Specification of a FrameImage facet 277
Specification of FrameImage data 277
Differences between monochrome and color 280
Sample unencoded FrameImage facet 281
Sample encoded FrameImage facet 282

Chapter 14: FrameVector Facet Format

Specification of a FrameVector facet 284
Specification of FrameVector data 284
Sample FrameVector facet 299

Chapter 15: Legal notices

Chapter 1: Introduction

MIF (Maker Interchange Format) is a group of ASCII statements that create an easily parsed, readable text file of all the text, graphics, formatting, and layout constructs that Adobe® FrameMaker® understands. Because MIF is an alternative representation of a FrameMaker document, it allows FrameMaker and other applications to exchange information while preserving graphics, document content, and format.

Why use MIF?

You can use MIF files to allow FrameMaker and other applications to exchange information. For example, you can write programs to convert graphics and text MIF and then import the MIF file into FrameMaker with the graphics and text intact. You can also save a FrameMaker document or book file as a MIF file and then write a program to convert the MIF file to another format. These conversion programs are called *filters*; filters allow you to convert FrameMaker document files into *foreign files* (files in another word processing or desktop publishing format), and foreign files into FrameMaker document files.

You can use MIF files with database publishing applications, which allow you to capture changing data from databases and format the data into high-quality documents containing both text and graphics information. You use the database to enter, manipulate, sort, and select data. You use FrameMaker to format the resulting data. You use MIF files as the data interchange format between the database and FrameMaker.

You can also use MIF files to do the following:

- Share documents with earlier versions of FrameMaker
- Perform custom document processing
- Set options for online documents in View Only format

These tasks are described in "Applications of MIF" on page 48. You can use other FrameMaker to perform some of these tasks. See "Other application tools" on page 52.

Using this manual

This manual:

- Describes the layout of MIF files.
- Provides a complete description of each MIF statement and its syntax.
- Provides examples of how to use MIF statements.
- Includes MIF statements for FrameMaker®.

To get the most from this manual you should be familiar with FrameMaker. For information about FrameMaker and its features, see the documentation for your product. In addition, if you are using MIF as an interchange format between FrameMaker and another application, you should be familiar with the tools needed to create and manipulate the other application, such as a programming language or database query language.

This chapter provides basic information about working with MIF files, including opening and saving MIF files in FrameMaker. It goes on to provide detailed information about the MIF language and its syntax.

For an introduction to writing MIF files, read , “Using MIF Statements.” You can then use the statement index, subject index, and table of contents to locate more specific information about a particular MIF statement.

For a description of a MIF statement, use the table of contents or statement index to locate the statement.

For a description of the differences between the MIF statements for this version of FrameMaker and earlier versions, see , “MIF Compatibility.”

Style conventions

This manual uses different fonts to represent different types of information.

- What you type is shown in

`text like this.`

- MIF statement names, pathnames, and filenames are also shown in

`text like this.`

- Placeholders (such as MIF data) are shown in

`text like this`.

- For example, the statement description for `PgfTag` is shown as:

`<PgfTag` *`tagstring`*`>`

- You replace *`tagstring`* with the tag of a paragraph format.

This manual also uses the term *FrameMaker*, (as in *FrameMaker document,* or *FrameMaker session*) to refer to FrameMaker and to refer to structured or unstructured documents.

Overview of MIF statements

When you are learning about MIF statements, you may find it useful to understand how FrameMaker represents documents.

How MIF statements represent documents

FrameMaker represents document components as *objects*. Different types of objects represent different components in a FrameMaker document. For example, a paragraph is considered an object; a paragraph format is considered a *formatting object*. The graphic objects that you create by using the Tools palette are yet another type of object.

Each object has *properties* that represent its characteristics. For example, a paragraph has properties that represent its left indent, the space above it, and its default font. A rectangle has properties that represent its width, height, and position on the page.

When FrameMaker creates a MIF file, it writes an ASCII statement for each object in the document or book. The statement includes substatements for the object’s properties.

For example, suppose a document (with no text frame) contains a rectangle that is 2 inches wide and 1 inch high. The rectangle is located 3 inches from the left side of the page and 1.5 inches from the top. MIF represents this rectangle with the following statement:

```
<Rectangle                          # Type of graphic object
                                    # Position and size: left offset, top offset,
                                    # width, and height
    <ShapeRect 3.0" 1.5" 2.0" 1.0">
>
```

FrameMaker also treats each document as an object and stores document preferences as properties of the document. For example, a document's page size and page numbering style are document properties.

FrameMaker documents have default objects

A FrameMaker document always has a certain set of default objects, formats, and preferences, even when you create a new document. When you create a MIF file, you usually provide the objects and properties that your document needs. However, if you don't provide all the objects and properties required in a FrameMaker document, the MIF interpreter fills in a set of default objects and document formats.

The MIF interpreter normally provides the following default objects:

- Predefined paragraph formats for body text, headers, and table cells
- Predefined character formats
- A right master page for single-sided documents and left and right master pages for double-sided documents
- A reference page
- Predefined table formats
- Predefined cross-reference formats
- Default pen and fill values and dash patterns for graphics
- Default colors
- Default document preferences, such as ruler settings
- Default condition tags

Although you can rely on the MIF interpreter to provide defaults, the exact properties and objects provided may vary depending on your FrameMaker configuration. The MIF interpreter uses default objects and properties that are specified in setup files and in templates. In UNIX® versions, these templates are `ASCIITemplate` and `NewTemplate`. You can modify these default objects and document formats by creating your own version of `ASCIITemplate` or `NewTemplate` or by modifying your setup files.

For more information about modifying the default templates and setup files, see the online manual *Customizing FrameMaker* for UNIX versions of FrameMaker. For the and Windows® version, see the chapter on templates in your user manual.

Current state and inheritance

FrameMaker has a MIF interpreter that reads and parses MIF files. When you open or import a MIF file, the interpreter reads the MIF statements and creates a FrameMaker document that contains the objects described in the MIF file.

When the interpreter reads a MIF file, it keeps track of the current state of certain objects. If the interpreter reads an object with properties that are not fully specified, it applies the current state to that object. When an object acquires the current state, it *inherits* the properties stored in that state.

For example, if the line width is set to 1 point for a graphic object, the interpreter continues to use a 1-point line width for graphic objects until a new value is specified in the MIF file. Similarly, if the MIF file specifies a format for a paragraph, the interpreter uses the same format until a new format is specified in the file.

The MIF interpreter keeps track of the following document objects and properties:

- Units
- Condition tag properties
- Paragraph format properties
- Character format properties

- Page properties
- Graphic frame properties
- Text frame properties
- Fill pattern
- Pen pattern
- Line width
- Line cap
- Line style (dash or solid)
- Color
- Text line alignment and character format

Because the interpreter also provides default objects for a document, the current state of an object may be determined by a default object. For example, if a document does not provide any paragraph formats, the interpreter applies a set of default paragraph properties to the first paragraph. Subsequent paragraphs use the same properties unless otherwise specified.

How FrameMaker identifies MIF files

A MIF file must be identified by a `MIFFile` or `Book` statement at the beginning of the file; otherwise FrameMaker simply reads the file as a text file. All other statements are optional; that is, a valid MIF file can contain only the `MIFFile` statement. Other document objects can be added as needed; FrameMaker provides a set of default objects if a MIF file does not supply them.

MIF statement syntax

The statement descriptions in this manual use the following conventions to describe syntax:

`<token data>`

`token data` where `token` represents one of the MIF statement names (such as `Pgf`) listed in the MIF statement descriptions later in this manual, and `data` represents one or more numbers, a string, a token, or nested statements. Markup statements are always delimited by angle brackets (<>); macro statements are not. For the syntax of macro statements, see "Macro statements" on page 56.

A *token* is an indivisible group of characters that identify a reserved word in a MIF statement. Tokens in MIF are case-sensitive. A token cannot contain white space characters, such as spaces, tabs, or newlines. For example, the following MIF statement is invalid because the token contains white space characters: `<Un its Uin>`

When the MIF interpreter finds white space characters that aren't part of the text of the document (as in the example MIF statement, `< Units  Uin  >`), it interprets the white space as token delimiters. When parsing the example statement, the MIF interpreter ignores the white space characters between the left angle bracket (<) and the first character of the token, `Units`. After reading the token, the MIF interpreter checks its validity. If the token is valid, the interpreter reads and parses the data portion of the statement. If the token is not valid, the interpreter ignores all text up to the corresponding right angle bracket (>), including any nested substatements. The interpreter then scans the file for the next left angle bracket that marks the beginning of the next MIF statement.

All statements, as well as all data portions of a statement, are optional. If you do not provide a data portion, the MIF interpreter assigns a default value to the statement.

Statement hierarchy

Some MIF statements can contain other statements. The contained statements are called *substatements*. In this manual, substatements are usually shown indented within the containing statements as follows:

```
<Document
    <DStartPage 1>
>
```

The indentation is not required in a MIF file, although it may make the file easier for you to read.

A MIF *main statement* appears at the *top level* of a file. A main statement cannot be nested within other statements. Some substatements can only appear within certain main statements.

The statement descriptions in this manual indicate the valid locations for a substatement by including it in all of the valid main statements. Main statements are identified in the statement description; for the correct order of main statements, see "MIF file layout" on page 53.

MIF data items

There are several general types of data items in a MIF statement. This manual uses the following terms and symbols to identify data items.

This term or symbol	Means
string	Left quotation mark (`), zero or more standard ASCII characters (you can also include UTF-8 characters), and a straight quotation mark ('). Example: `ab cdef ghij'
tagstring	A string that names a format tag, such as a paragraph format tag. A *tagstring* value must be unique; case is significant. A statement that refers to a *tagstring* must exactly match the *tagstring* value. A *tagstring* value can include any character from the FrameMaker character set.
pathname	A string specifying a pathname (see "Device-independent pathnames" on page 7).
boolean	A value of either Yes or No. Case is significant.
integer	Integer whose range depends on the associated statement name.
ID	Integer that specifies a unique ID. An ID can be any positive integer between 1 and 65535, inclusive. A statement that refers to an ID must exactly match the ID.
dimension	Decimal number signifying a dimension. You can specify the units, such as 1.11", 72 pt, and 8.3 cm. If no units are specified, the default unit is used.
degrees	A decimal number signifying an angle value in degrees. You cannot specify units; any number is interpreted as a degree value.
percentage	A decimal number signifying a percentage value. You cannot specify units; any number is interpreted as a percentage value.
metric	A dimension specified in units that represent points, where one point is 1/72 inch (see "Math values" on page 6). Only used in MathFullForm statements.
W H	Pair of dimensions representing width and height. You can specify the units.
X Y	Coordinates of a point. Coordinates originate at the upper-left corner of the page or graphic frame. You can specify the units.
L T R B	Coordinates representing left, top, right, and bottom indents. You can specify the units.
L T W H	Coordinates representing the left and top indents plus the dimensions representing the width and height of an object. You can specify the units.

This term or symbol	Means
X Y W H	Coordinates of a point on the physical screen represented by *X* and *Y* plus dimensions describing the width and height. Used only by the `DWindowRect` and `DViewRect` statements within the `Document` statement and the `BWindowRect` statement within the `Book` statement. The values are in pixels; you cannot specify the units.
keyword	A token value. The allowed token values are listed for each statement; you can provide only one value.
<token…>	Ellipsis points in a statement indicate required substatements or arguments. The entire expanded statement occurs at this point.

Unit values

You can specify the unit of measurement for most dimension data items. The following table lists the units of measurement that FrameMaker supports and their notation in MIF.

Measurement unit	Notation in MIF	Relationship to other units
point	`pt` or `point`	1/72 inch
inch	" or `in`	72 points
millimeter	`mm` or `millimeter`	1 inch is 25.4 mm
centimeter	`cm or centimeter`	1 inch is 2.54 cm
pica	`pc or pica`	12 points
didot	`dd or didot`	0.01483 inches
cicero	`cc or cicero`	12 didots
pixel	`px`	.0625 pica

Dimension data types can mix different units of measurement. For example, the statement `<CellMargins L T R B>` can be written as either of the following:

```
<CellMargins 6 pt 18 pt 6 pt 24 pt>
<CellMargins 6 pt .25" .5 pica 2 pica>
```

Math values

The `MathFullForm` statement uses *`metric`* values in formatting codes. A *`metric`* unit represents one point (1/72 inch). The *`metric`* type is a 32-bit fixed-point number. The 16 most significant bits of a *`metric`* value represent the digits before the decimal; the 16 least significant bits represent the digits after the decimal. Therefore, 1 point is expressed as hexadecimal `0x10000` or decimal `65536`. The following table shows how to convert *`metric`* values into equivalent measurement units.

To get this unit	Divide the metric value by this number
point	65536
inch	4718592
millimeter	185771
centimeter	1857713
pica	786432
didot	6997

To get this unit	Divide the metric value by this number
cicero	839724
pixel	49152

Character set in strings

MIF string data uses the FrameMaker character set (see the *Quick Reference* for your FrameMaker product). MIF strings must begin with a left quotation mark (ASCII character code `0x60`) and end with a straight quotation mark (ASCII character code `0x27`). Within a string, you can include any character in the FrameMaker character set. However, because a MIF file can contain only standard ASCII characters and because of MIF parsing requirements, you must represent certain characters with backslash (\) sequences.

Character	Representation
Tab	\t
>	\>
'	\q
`	\Q
\	\\
nonstandard ASCII	\x*nn*

Note: *The \xnn character is supported only for legacy MIF files.*

All FrameMaker characters with values above the standard ASCII range (greater than `\x7f`) are represented in a string by using `\xnn` notation, where *nn* represents the hexadecimal code for the character. The hexadecimal digits must be followed by a space.

When using special characters in a variable definition, you can also use a hexadecimal notation or Unicode notation. In the previous example, the hexadecimal notation for the paragraph symbol (¶) is \xa6. Alternatively, you can use the \u00B6 Unicode notation to represent the same character.

The following example shows a FrameMaker document line and its representation in a MIF string.

In a FrameMaker document	In MIF
Some `symbols': > \Ø¿!	`Some \Qsymbols\q: \> \\Ø¿!'

You can also use the `Char` statement to include certain predefined special characters in a `ParaLine` statement (see "Char statement" on page 134).

Device-independent pathnames

Several MIF statements require pathnames as values. You should supply a device-independent pathname so that files can easily be transported across different system types. Because of MIF parsing requirements, you must use the following syntax to supply a pathname:

`` `<code\>name<code\>name<code\>name… ' ``

where *name* is the name of a component in the file's path and *code* identifies the role of the component in the path. The following table lists codes and their meanings.

Code	Meaning
r	Root of UNIX file tree (UNIX only)
v	Volume or drive (Windows)
h	Host (Apollo only)
c	Component
u	Up one level in the file tree

When you specify a device-independent pathname in a MIF string, you must precede any right angle brackets (>) with backslashes (\), as shown in the syntax above.

Absolute pathnames

An *absolute pathname* shows the location of a file beginning with the root directory, volume, or drive. The following table specifies device-independent, absolute pathnames for the different versions of FrameMaker.

In this version	The pathname appears as this MIF string
UNIX	`` `<r\><c\>MyDirectory<c\>MySubdirectory<c\>Filename' ``
Windows	`` `<v\>c:<c\>mydir<c\>subdir<c\>filename' ``

Relative pathnames

A *relative pathname* shows the location of a file relative to the current directory. In all FrameMaker versions, the device-independent, relative pathname for the same file is:

```
`<c\>Filename'
```

Chapter 2: Using MIF Statements

MIF statements can completely describe any Adobe® FrameMaker® document, no matter how complex. As a result, you often need many MIF statements to describe a document. To learn how to use MIF statements, it helps to begin with some simple examples.

This chapter introduces you to MIF, beginning with a simple MIF example file with only a few lines of text. Additional examples show how to add common document objects, such as paragraph formats, a table, and a custom page layout, to this simple MIF file.

The examples in this chapter are also provided in online sample files. You can open these examples in FrameMaker and experiment with them by adding additional MIF statements. Look for the sample files in the following location:

In this version	Look here
UNIX	`$FMHOME/fminit/`*`language`*`/Samples/MIF`, where *`language`* is the language in use, such as `usenglish`
Windows	The `MIF` directory under the `samples` directory

Working with MIF files

A MIF file is an alternate representation of a FrameMaker document in ASCII format. MIF files are usually generated by FrameMaker or by an application that writes out MIF statements. You can, however, create MIF files by using a text editor or by using FrameMaker as a text editor. This section provides some general information about working with MIF files regardless of the method you use to create them.

Opening and saving MIF files

When you save a FrameMaker document, you usually save it in Normal format, FrameMaker's binary format for document files. To save a document as a MIF file, choose Save As from the File menu. In the Save Document dialog box, choose Interchange (MIF) from the Format pop-up menu. You should give the saved file the suffix `.mif` to distinguish it from a file saved in binary format.

When you open or import a MIF file, FrameMaker reads the file directly, translating it into a FrameMaker document or book. When you save the document in Normal format, FrameMaker creates a binary document file. To prevent overwriting the original MIF file, remove the `.mif` file suffix and replace it with a different suffix (or no suffix).

If you use FrameMaker to edit a MIF file, you must prevent it from interpreting MIF statements when you open the file by holding down a modifier key and clicking Open in the Open dialog box.

In this version	Use this modifier key
UNIX	Shift
Windows	Control or Shift

Save the edited MIF file as a text file by using the Save As command and choosing Text Only from the Format pop-up menu. Give the saved file the suffix `.mif`. When you save a document as Text Only, FrameMaker asks you where to place carriage returns. For a MIF file, choose the Only between Paragraphs option.

In UNIX versions, FrameMaker saves a document in text format in the ISO Latin-1 character encoding. You can change the character encoding to ASCII by changing the value of an X resource. See the description of character encoding in the online manual *Customizing FrameMaker*. In the Windows version, press Esc F t c to toggle between FrameMaker's character encoding and ANSI for Windows.

Importing MIF files

You can use the File menu's Import>File command to import MIF files into an existing document, but you must make sure that the imported statements are valid at the location where you are importing them. A MIF file can describe both text and graphics; make sure that you have selected either a place in the text flow (if you are importing text or graphics) or an anchored frame (if you are importing graphics).

For example, to import a MIF file that describes a graphic, first create an anchored frame in a document, select the frame, and then import the MIF file (see "Bar chart example" on page 232).

When you import or include MIF files, make sure that object IDs are unique in the final document and that references to object IDs are correct (see "Generic object statements" on page 112). The object IDs must be unique for all objects (TextRect, TblId, Group, and AFrame use the ID for identification) in the document.

Editing MIF files

You normally use a text editor to edit a MIF file. If you use FrameMaker to enter text into a MIF file, be sure to open the MIF file as a text file and turn off Smart Quotes. If you leave Smart Quotes on, you must use a key sequence to type the quotation marks that enclose a MIF string (`'). To enter a left quotation mark, type Control-`. To enter a straight quotation mark, type Control-'.

Although MIF statements are usually generated by a program, while you learn MIF or test and debug an application that generates MIF, you may need to manually generate MIF statements. In either case, you can minimize the number of MIF statements that your application needs to generate or that you need to type in.

The following suggestions may be helpful when you are working with MIF statements:

- Edit a MIF file generated by FrameMaker.
- You can edit a MIF file generated by FrameMaker or copy a group of statements from a MIF file into your file and then edit the statements. An easy way to use FrameMaker to generate a MIF file is to create an empty document by using the New command and then saving it as a MIF file.
- Test one object at a time.
- While testing an object in a document or learning about the MIF statements that describe an object, work with just that object. For example, if you work with a document that contains both tables and anchored frames, start by creating the MIF statements that describe tables. Then add the statements that describe anchored frames.
- Use the default properties provided by FrameMaker.
- If you are not concerned with testing certain document components, let FrameMaker provide a set of default document objects and formats.

MIF file layout

FrameMaker writes the objects in a MIF document file in the following order:

This section	Contains these objects
File ID	MIF file identification line (`MIFFile` statement)
Units	Default units (`Units` statement)

This section	Contains these objects
Catalogs	Color Condition Paragraph Format Element Font or Character Format Ruling Table Format Views
Formats	Variable Cross-reference
Objects	Document Dictionary Anchored frames Tables Pages Text flows

FrameMaker provides all of these objects, even if the object is empty. To avoid unpredictable results in a document, you must follow this order when you create a MIF file.

Creating a simple MIF file for FrameMaker

Note: *The rest of this chapter explains how to create some simple MIF files for FrameMaker by hand. These instructions do not apply to structured documents, which require that you create elements first.*

The most accurate source of information about MIF files is a MIF file generated by FrameMaker. MIF files generated by FrameMaker can be very lengthy because FrameMaker repeats information and provides default objects and formats for all documents. You may find it difficult to determine the minimum number of statements that are necessary to define your document by looking at a FrameMaker-generated MIF file.

To better understand how FrameMaker reads MIF files, study the following example. This MIF file uses only four statements to describe a document that contains one line of text:

```
<MIFFile 2015>                          # The only required statement
<Para                                   # Begin a paragraph
    <ParaLine                           # Begin a line within the paragraph
        <String `Hello World'># The actual text of this document
    >                                   # end of Paraline #End of ParaLine statement
>                                       # end of Para #End of Para statement
```

The `MIFFile` statement is required in each MIF file. It identifies the FrameMaker version and must appear on the first line of the file. All other statements are optional; that is, FrameMaker provides a set of default objects if you specify none.

Comments in a MIF file are preceded by a number sign (#). By convention, the substatements in a MIF statement are indented to show their nesting level and to make the file easier to read. The MIF interpreter ignores spaces at the beginning of a line.

This example is in the sample file `hello.mif`. To see how FrameMaker provides defaults for a document, open this file in FrameMaker. Even though the MIF file does not specify any formatting, FrameMaker provides a default Paragraph Catalog and Character Catalog. In addition, it provides a right master page, as well as many other default properties.

Save this document as a MIF file and open the FrameMaker-generated MIF file in a text editor or in FrameMaker as a text file. (For information on how to save and open MIF files, see "Opening and saving MIF files" on page 9.)

You'll see that the MIF interpreter has taken the original 6-line file and generated over 1,000 lines of MIF statements that describe all the default objects and their properties. To see the actual text of the document, go to the end of the file.

This example demonstrates an important point about MIF files. Your MIF file can be very sparse; the MIF interpreter supplies missing information. Most documents are not this simple, however, and require some formatting. The following sections describe how to add additional document components, such as paragraph and character formats, a table, and custom page layouts, to this minimal MIF file.

Creating and applying paragraph formats

In a FrameMaker document, paragraphs have formatting properties that specify the appearance of the paragraph's text. A paragraph format includes the font family and size, indents, tab stops, the space between lines in a paragraph, the space before and after a paragraph, and the direction of the text. The text direction can be either left to right for languages like English and German, or right to left for languages like Arabic and Hebrew. In a FrameMaker document, the end of a paragraph is denoted by a single carriage return. You control the amount of space above and below the paragraph by modifying the paragraph's format, not by adding extra carriage returns.

In a FrameMaker document, you store paragraph formats in a Paragraph Catalog and assign a *tag* (name) to the format. You can then apply the same format to many paragraphs by assigning the format tag to the paragraphs. You can also format a paragraph individually, without storing the format in the Paragraph Catalog. Or, you can assign a format from the Paragraph Catalog and then override some of the properties within a particular paragraph. Formats that are not stored in the Paragraph Catalog are called *local formats*.

Creating a paragraph

In a MIF file, paragraphs are defined by a `Para` statement. A `Para` statement contains one or more `ParaLine` statements that contain the lines in a paragraph; the actual text of the line is enclosed in one or more `String` statements:

```
<Para                                   # Begin a paragraph
    <ParaLine                           # Begin a line within the paragraph
        <String `Hello World'># The actual text of this document
    >                                   # End of ParaLine statement
>                                       # End of Para statement
```

The `Para`, `ParaLine`, and `String` statements are the only required statements to import text. You could use this example to import a simple document into FrameMaker by placing each paragraph in a `Para` statement. Break the paragraph text into a series of `String` statements contained in one `ParaLine` statement. It doesn't matter how you break up text lines within a `Para` statement; the MIF interpreter automatically wraps lines when it reads the MIF file.

Some characters must be represented by backslash sequences in a MIF string. For more information, see "Character set in strings" on page 7.

Creating a paragraph format

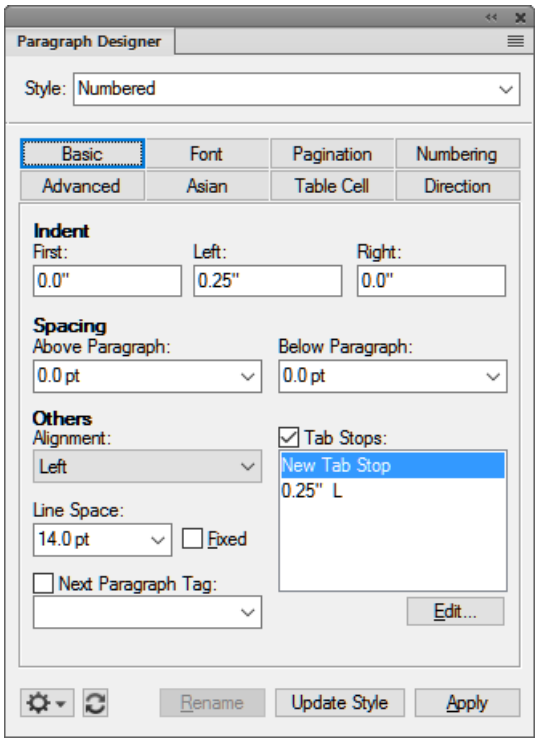
Within a FrameMaker document, you define a paragraph format by using the Paragraph Designer to specify the paragraph's properties. In a MIF file, you define a paragraph format by using the `Pgf` statement.

The `Pgf` statement contains a group of substatements that describe all of a paragraph's properties. It has the following syntax:

```
<Pgf
    <property value>
    <property value>
    ...
>
```

A `Pgf` statement is quite long, so learning how to relate its substatements to the paragraph's properties may take some practice. Usually a MIF statement name is similar to the name of the setting within a dialog box. The following examples show the property dialog boxes from the Paragraph Designer with the related `Pgf` substatements.

Suppose you have created a paragraph format for a numbered list item with Basic properties defined as follows in the Paragraph Designer.



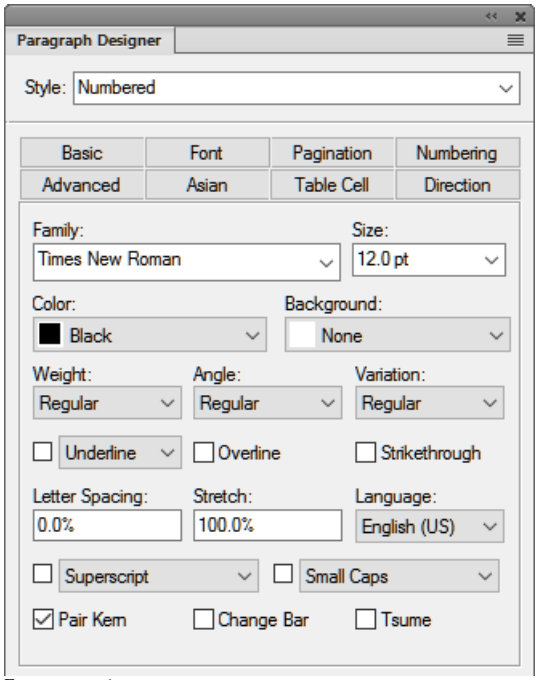
Basic properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfTag `Numbered'>`	Paragraph Tag
`<PgfFIndent 0.0">`	First Indent
`<PgfLIndent 0.25">`	Left Indent
`<PgfRIndent 0.0">`	Right Indent
`<PgfAlignment Left >`	Alignment
`<PgfSpBefore 0.0 pt>`	Space Above ¶

In MIF file	In Paragraph Designer
<PgfSpAfter 0.0 pt>	Space Below ¶
<PgfLeading 2.0 pt>	Line Spacing (leading is added to font size)
<PgfLineSpacing Fixed>	Line Spacing (fixed)
<PgfNumTabs 1>	Number of tab stops
<TabStop	Begin definition of tab
<TSX 0.25">	Tab position
<TSType Left >	Tab type
<TSLeaderStr `'>	Tab leader (none)
> # end of TabStop	
<PgfUseNextTag No >	Turn off Next ¶ Tag feature
<PgfNextTag `'>	Next ¶ Tag name (none)

The Default Font properties are defined as follows in the Paragraph Designer.



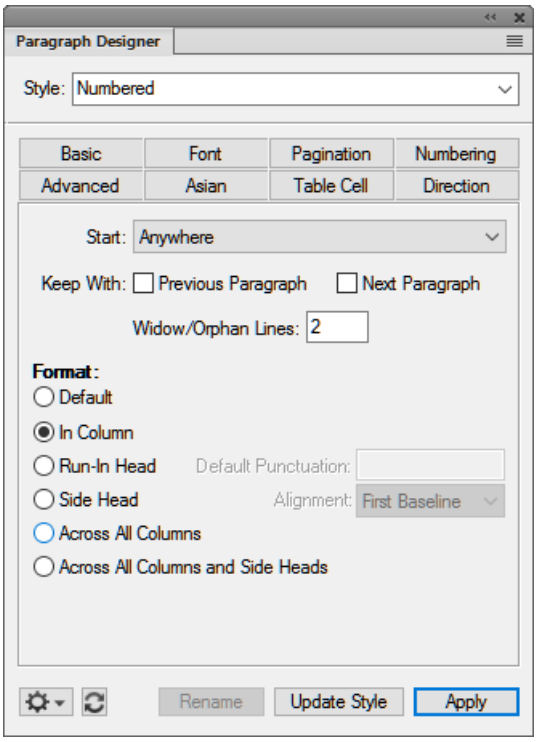
Font properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
<PgfFont	
<FFamily `Times'>	Family

In MIF file	In Paragraph Designer
<FSize 12.0 pt>	Size
<FEncoding>	
<FAngle `Regular'>	Angle
<FWeight `Regular'>	Weight
<FLanguage>	Language
<FVar `Regular'>	Variation
<FColor `Black'>	Color
<FDW 0.0 pt>	Spread
<FStretch 100%>	Stretch
<FUnderlining NoUnderlining >	Underline
<FOverline No >	Overline
<FStrike No >	Strikethrough
<FChangeBar No >	Change Bar
<FPosition FNormal >	Superscript/Subscript
<FCase FAsTyped >	Capitalization
<FPairKern Yes >	Pair Kern
<FTsume No>	Tsume (Asian systems only)
> # end of PgfFont	

The Pagination properties are defined as follows in the Paragraph Designer.

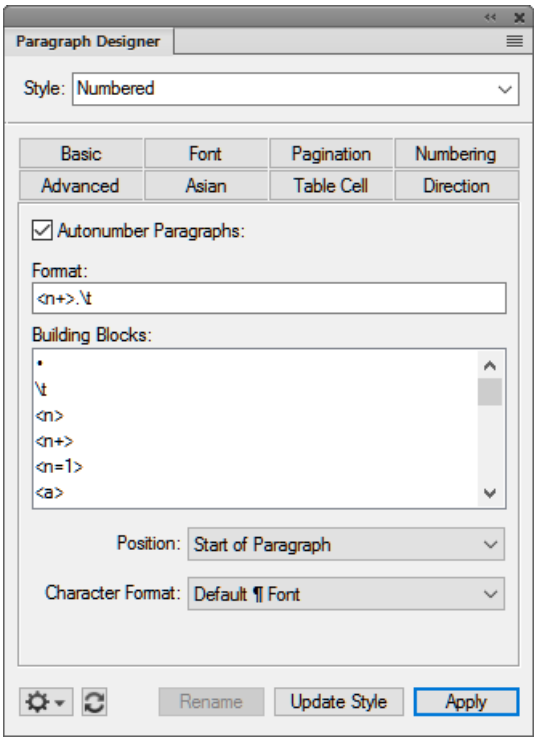


Pagination properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfPlacement Anywhere >`	Start
`<PgfWithNext No >`	Keep With Next Pgf
`<PgfWithPrev No >`	Keep With Previous Pgf
`<PgfBlockSize 1>`	Widow/Orphan Lines
`<PgfPlacementStyle Normal >`	Format (paragraph placement)
`<PgfRunInDefaultPunct `. '>`	Run-in Head Default Punctuation (a period followed by an em space)

The Numbering properties are defined as follows in the Paragraph Designer.

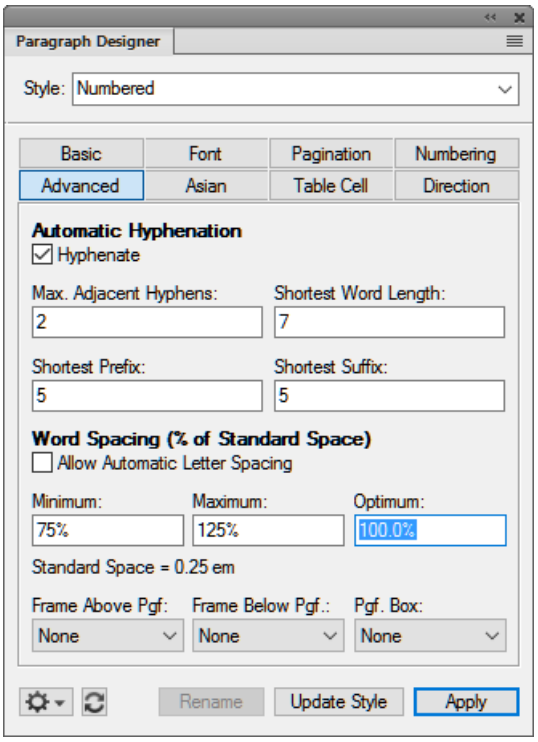


Numbering properties

The following table shows the corresponding MIF statements:

In MIF file	**In Paragraph Designer**
`<PgfAutoNum Yes >`	Turn on Autonumber
`<PgfNumFormat `<n+\>.\\t' >`	Autonumber Format (a number followed by a period and a tab)
`<PgfNumberFont `' >`	Character Format (Default ¶ Format)
`<PgfNumAtEnd No >`	Position (Start of Paragraph)

The Advanced properties are defined as follows in the Paragraph Designer.

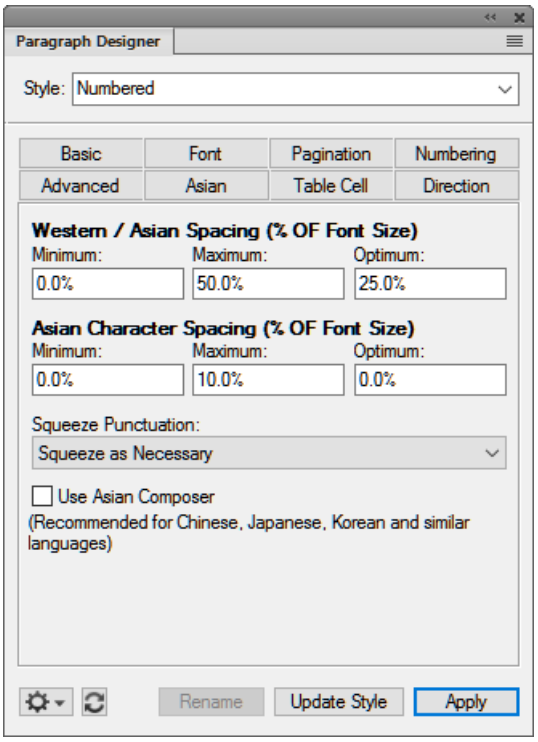


Advanced properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfHyphenate Yes >`	Automatic Hyphenation (on)
`<HyphenMaxLines 2>`	Max. # Adjacent
`<HyphenMinWord 5>`	Shortest Word
`<HyphenMinPrefix 3>`	Shortest Prefix
`<HyphenMinSuffix 3>`	Shortest Suffix
`<PgfMinWordSpace 90>`	Minimum Word Spacing
`<PgfOptWordSpace 100>`	Optimum Word Spacing
`<PgfMaxWordSpace 110>`	Maximum Word Spacing
`<PgfLetterSpace Yes >`	Allow Automatic Letter Spacing
`<PgfTopSeparator `'>`	Frame Above ¶
`<PgfBotSeparator `'>`	Frame Below ¶

The Asian properties are defined as follows in the Paragraph Designer.



Asian properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfMinJRomanLetterSpace` *percentage*`>`	Minimum (Western/Asian Spacing)
`<PgfOptJRomanLetterSpace` *percentage*`>`	Optimum (Western/Asian Spacing)
`<PgfMaxJRomanLetterSpace` *percentage*`>`	Maximum (Western/Asian Spacing)
`<PgfMinJLetterSpace` *percentage*`>`	Minimum (Asian Character Spacing)
`<PgfOptJLetterSpace` *percentage*`>`	Optimum (Asian Character Spacing)
`<PgfMaxJLetterSpace` *percentage*`>`	Maximum (Asian Character Spacing)
`<PgfYakumonoType` *string*`>`	Asian Punctuation

The Table Cell properties are defined as follows in the Paragraph Designer.

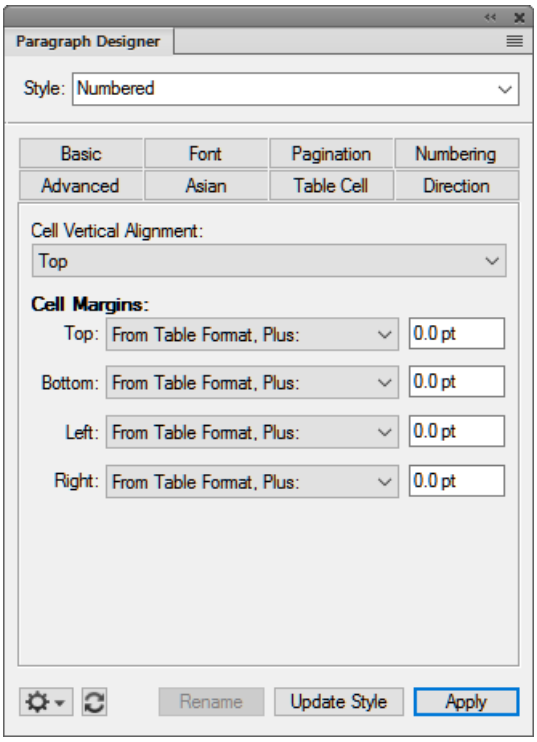
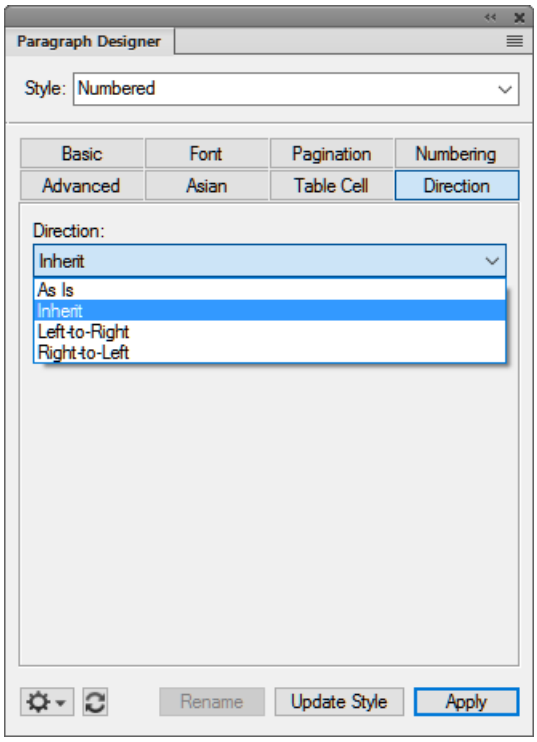


Table cell properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfCellAlignment Top >`	Cell Vertical Alignment
`<PgfCellMargins 0.0 pt 0.0 pt 0.0 pt 0.0 pt>`	Cell Margins
`<PgfCellTMarginFixed No >`	Top
`<PgfCellBMarginFixed No >`	Bottom
`<PgfCellLMarginFixed No >`	Left
`<PgfCellRMarginFixed No >`	Right

The Direction properties are defined as follows in the Paragraph Designer.



Direction properties

The following table shows the corresponding MIF statements:

In MIF file	In Paragraph Designer
`<PgfDir LTR>`	Direction of the paragraph text
`> # end of Pgf`	

Adding a Paragraph Catalog

In a MIF file, you define a Paragraph Catalog by using a `PgfCatalog` statement. The Paragraph Catalog contains one or more paragraph formats, which are defined by `Pgf` statements. A `PgfCatalog` statement looks like this:

```
<PgfCatalog
    <Pgf…>                          # A paragraph format description
    <Pgf…>                          # More paragraph formats
>                                   # end of PgfCatalog
```

The `Pgf` statement describes a complete paragraph format. For example, the sample file `pgfcat.mif` stores the paragraph format 1Heading in the Paragraph Catalog:

```
<MIFFile 2015>                      # Hand generated
<PgfCatalog
    <Pgf
        <PgfTag `1Heading'>
        <PgfUseNextTag Yes >
        <PgfNextTag `Body'>
        <PgfAlignment Left >
```

```
        <PgfFIndent  0.0">
        <PgfLIndent  0.0">
        <PgfRIndent  0.0">
        ...
        <PgfBoxColor NoColor>
        <PgfAsianComposer No>
        <PgfDir LTR>
    >                               # end of Pgf
>                                   # end of PgfCatalog
```

If you open `pgfcat.mif` in FrameMaker, you'll see that the Paragraph Catalog contains a single paragraph format called 1Heading. If you supply a Paragraph Catalog, the paragraph formats in your catalog replace those in the default catalog; they do not supplement the default formats.

If you do not supply a Paragraph Catalog in a MIF file, the MIF interpreter provides a default Paragraph Catalog with predefined paragraph formats.

If a `Pgf` statement provides only the name of a paragraph format, the MIF interpreter supplies default values for the rest of the paragraph properties when it reads in the MIF file.

Applying a paragraph format

To apply a format from the Paragraph Catalog to a paragraph, use the `PgfTag` statement to include the format tag name within the `Para` statement. For example, to apply the previously defined format 1Heading to a paragraph, use the following statements:

```
<Para
    <PgfTag `1Heading'>
    <ParaLine
        <String `This line has the format called 1Heading.'>
    >                               # end of ParaLine
>                                   # end of Para
```

To apply a format from the Paragraph Catalog and then locally override some properties, use a partial `Pgf` statement within the `Para` statement. The following MIF example applies the paragraph format 1Heading, then changes the alignment:

```
<Para
    <PgfTag `1Heading'>
    <Pgf
        <PgfAlignment Center>
    >                               # end of Pgf
    <ParaLine
        <String `This line is centered.'>
    >                               # end of ParaLine
>                                   # end of Para
```

To locally define a paragraph format, include a complete `Pgf` statement within the `Para` statement:

```
<Para
    <Pgf
        <PgfTag `2Heading'>
        <PgfUseNextTag Yes >
        <PgfNextTag `Body'>
        <PgfAlignment Left >
        <PgfFIndent  0.0">
        <PgfLIndent  0.0">
        ...
    >                               # end of Pgf
    <ParaLine
        <String `A locally formatted heading'>
    >                               # end of ParaLine
>                                   # end of Para
```

For a complete description of Pgf property statements, see page 62.

How paragraphs inherit properties

Paragraphs can inherit properties from other paragraphs in a MIF file. If a Pgf statement does not provide values for each paragraph property, it acquires any property values explicitly defined in a previous Pgf statement. Because the MIF interpreter sequentially reads MIF files, it uses the most recently defined Pgf statement that occurs before the current statement in the file.

For example, the following MIF code applies the default format named Body to the first paragraph in a document and locally overrides the paragraph font:

```
<Para
    <Pgf
        <PgfTag `Body'>
        <PgfFont
            <FWeight `Bold'>
        >                           # end of PgfFont
    >                               # end of Pgf
    <ParaLine
        <String `First paragraph in document.'>
    >                               # end of ParaLine
>                                   # end of Para
<Para
    <ParaLine
        <String `Second paragraph in document.'>
    >                               # end of ParaLine
>                                   # end of Para
```

The previous example is in the sample file pgffmt.mif. If you open this file in FrameMaker, you'll find that the second paragraph also has the new font property.

A paragraph property remains in effect until the property value is changed by a subsequent MIF statement. To change a paragraph property to another state, supply a Pgf statement containing the paragraph property statement set to the new state.

Thus, in the previous example, you could change the font from Bold to Regular in a Pgf statement in the second Para statement:

```
<Para
    <Pgf
        <PgfFont
            <FWeight `Regular'>
        >                           # end of PgfFont
    >                               # end of Pgf
    <ParaLine
        <String `Second paragraph in document.'>
    >                               # end of ParaLine
>                                   # end of Para
```

To summarize, paragraphs inherit formats as follows:

- Formats in the Paragraph Catalog inherit properties from the formats above them.
- Locally defined paragraph formats inherit properties from previously specified formats.
- Text lines in anchored frames inherit font properties from previously specified formats, including the last format in the Paragraph Catalog and previous text lines.

Tips

The following hints may help you minimize the MIF statements for paragraph formats:

- If possible, use the formats in the default Paragraph Catalog (don't supply a `PgfCatalog` statement). If you know the names of the default paragraph formats, you can tag paragraphs with the `PgfTag` statement.
- If you know that a document will use a particular template when it is imported into a FrameMaker document, you can just tag the paragraphs in the text flow. Don't create a new Paragraph Catalog in MIF; it's easier to create catalogs in FrameMaker document templates.
- If you need to provide a full Paragraph Catalog in a MIF file, you can still use FrameMaker to ease the task of creating a catalog. Create a template in FrameMaker, save the template as a MIF file, and include the Paragraph Catalog in your document. For instructions, see "Including template files" on page 45.

Creating and applying character formats

You can define character formats locally or store them in the Character Catalog and apply the formats to text selections. Creating and applying character formats is very similar to creating and applying paragraph formats as described in the previous section. Because the two methods are similar, this section just summarizes how to create and apply character formats.

In a MIF file, the Character Catalog is contained in a `FontCatalog` statement. The `FontCatalog` statement contains named character formats in a list of `Font` statements. A `FontCatalog` statement looks like this:

```
<FontCatalog
    <Font...>                       # Describes a character format
    <Font...>                       # Describes a character format
>                                   # end of FontCatalog
```

A `Font` statement specifies the properties of a character format; these are the same properties specified in the Character Designer. The `Font` statement is just like the `PgfFont` statement that you use to define the default font in a paragraph format. See "PgfFont and Font statements" on page 67 for a complete description of a `Font` statement.

To apply a predefined character format to text, use the `FTag` statement:

```
<MIFFile 2015>                      # Hand generated
<FontCatalog
    <Font
        <FTag `Emphasis'>
        <FAngle `Italic'>
    >                               # end of Font
>                                   # end of FontCatalog
<Para
    <PgfTag `Body'>
    <ParaLine
        <String `You can format characters within a paragraph by '>
        <Font
            <FTag `Emphasis'>
        >                           # end of Font
        <String `applying'>
        <Font
            <FTag `'>
        >                           # end of Font
        <String ` a character format from the character catalog.'>
    >                               # end of ParaLine
>                                   # end of Para
```

Remember to include a second `Font` statement to end the scope of the applied character format.

To locally define a character format, use a complete `Font` statement:

```
<Para
    <PgfTag `Body'>
```

```
    <ParaLine
        <String `You can also format characters by '>
        <Font
            <FTag `Emphasis'>
        …character property statements…
        >                           # end of Font
        <String `applying'>
        <Font
            <FTag `'>
        >                           # end of Font
        <String ` a locally defined character format.'>
    >                               # end of ParaLine
>                                   # end of Para
```

Like paragraph formats, character formats inherit properties from previously defined character formats. Unlike paragraph formats, however, a character format ends at the close of a `Para` statement.

See the sample file `charfmt.mif` for examples of using character formats.

Creating and formatting tables

You can create tables in FrameMaker documents, edit them, and apply table formats to them. Tables can have heading rows, body rows, and footing rows. Each row consists of table cells that contain the actual contents of the table.

Table 1: Coffee Inventory — Title

Coffee	Bags	Status	Price per bag
Brazil Santos	50	Prompt	$455.00
Celebes Kalossi	29	In Stock	$924.00
Colombian	25	In Stock	$474.35
			$1,853.35

Heading row

Body rows

Footing row

Tables are like paragraphs in that they have a format. A table format controls the appearance of a table, including the number and width of columns, the types of ruling or shading in rows and columns, and the table's position in a text column. Table formats can be named, stored in a Table Catalog, and applied to many tables. A table format can also be defined locally.

In a FrameMaker document, tables appear where they have been placed in the text flow. A table behaves like an anchored frame, so a table flows with the surrounding text unless you give it a specific location. In a MIF file, the document's tables are collected in one place and a placeholder for each table indicates the table's position in the text flow.

You create a table in a MIF file as follows:

- Specify the contents of the table by using a `Tbl` statement. An individual table is called a *table instance*. All table instances are stored in one `Tbls` statement. Assign each table instance a unique ID number.
- Indicate the position of the table in the text flow by using an `ATbl` statement. The `ATbl` statement is the placeholder, or *anchor*, for the table instance. It refers to the table instance's unique ID.

- Specify the table format by using a `TblFormat` statement. Formats can be named and stored in the Table Catalog, which is defined by a `TblCatalog` statement, or locally defined within a table.

Creating a table instance

All table instances in a document are contained in a `Tbls` statement. The `Tbls` statement contains a list of `Tbl` statements, one for each table instance. A document can have only one `Tbls` statement, which must occur before any of the table anchors in the text flow.

The `Tbl` statement contains the actual contents of the table cells in a list of MIF substatements. Like other MIF statements, this list can be quite long. The following is a template for a `Tbl` statement:

```
<Tbl
    <TblID…>                        # A unique ID for the table
    <TblFormat…>                    # The table format
    <TblNumColumns…>                # Number of columns in this table--required
    <TblColumnWidth…>               # Column width, one for each column
    <TblH                           # The heading; omit if no heading
        <Row                        # One Row statement for each row
            <Cell…>                 # One statement for each cell in the row
        >                           # end of Row
    <TblBody                        # The body of the table
        <Row…>                      # One for each row in body
    >                               # end of TblBody
    <TblF                           # The footer; omit if no footer
        <Row…>                      # One for each row in footer
    >                               # end of TblF
>                                   # end of Tbl
```

The `TblID` statement assigns a unique ID to the table instance. The `TblFormat` statement provides the table format. You can use the `TblFormat` statement to apply a table format from the Table Catalog, apply a format from the catalog and override some of its properties, or completely specify the table format locally. Because the tables in a document often share similar characteristics, you usually store table formats in the Table Catalog. Table instances can always override the applied format.

The `TblNumColumns` statement specifies the number of columns in the table instance. It is required in every table.

The `TblH`, `TblBody`, and `TblF` statements contain the table heading, body, and footer rows. If a table does not have a heading or footing, omit the statements.

Here's an example of a simple table that uses a default format from the Table Catalog. The table has one heading row, one body row, and no footing rows:

Coffee	Price per Bag
Brazil Santos	$455.00

You can use the following MIF statements to create this simple table:

```
<MIFFile 2015>
<Tbls
    <Tbl
        <TblID 1>                       # ID for this table
        <TblTag `Format A'>             # Applies format from Table Catalog
        <TblNumColumns 2>               # Number of columns in this table
        <TblColumnWidth  2.0">          # Width of first column
        <TblColumnWidth  1.5">          # Width of second column
        <TblH                           # Begin table heading
            <Row                        # Begin row
```

```
                <Cell                       # First cell in row
                    <CellContent
                        <Para               # Cells can contain paragraphs
                            <PgfTag `CellHeading'># Applies format from Paragraph Catalog
                            <ParaLine
                                <String `Coffee'># Text in this cell
                            >
                        >                   # end of Para
                    >                       # end of CellContent
                >                           # end of Cell
                <Cell                       # Second cell in row
                    <CellContent
                        <Para
                            <PgfTag `CellHeading'>
                                <ParaLine
                                 <String `Price per Bag'>
                            >
                        >                   # end of Para
                    >                       # end of CellContent
                >                           # end of Cell
            >                               # end of Row
        >                                   # end of TblH
    <TblBody                                # Table body
        <Row                                # Begin row
            <Cell                           # First cell in row
                <CellContent
                    <Para
                        <PgfTag `CellBody'>
                        <ParaLine
                            <String `Brazil Santos'>
                        >
                    >                       # end of Para
                >                           # end of CellContent
            >                               # end of Cell
            <Cell                           # Second cell in row
                <CellContent
                    <Para
                        <PgfTag `CellBody'>
                        <ParaLine
                            <String `$455.00'>
                        >
                    >                       # end of Para
                >                           # end of CellContent
            >                               # end of Cell
        >                                   # end of Row
    >                                       # end of TblBody
>                                           # end of Tbl
>                                           # end of Tbls
```

A table cell is a text column that contains an untagged text flow not connected to any other flows. You can put any kind of text or graphics in a table cell. The cell automatically grows vertically to accommodate the inserted text or graphic; however, the width of the column remains fixed.

Adding a table anchor

To indicate the position of a table in the text flow, you must add an `ATbl` statement. The `ATbl` statement refers to the unique ID specified by the `TblID` statement in the table instance. For example, to insert the table defined in the previous example, you would add the following statements to the minimal MIF file:

```
<Para
```

```
    <ParaLine
        <String `Coffee prices for January'>
        <ATbl 1>                        # Matches table ID in Tbl statement
    >                                   # end of ParaLine
>                                       # end of Para
```

This example is in the sample file `table.mif`. If you open this file in FrameMaker, you'll see that the anchor symbol for the table appears at the end of the sentence. To place the table anchor between two words in the sentence, use the following statements:

```
<Para
    <ParaLine
        <String `Coffee prices '>
        <ATbl 1>
        <String `for January'>
    >                                   # end of ParaLine
>                                       # end of Para
```

Note that the `ATbl` statement appears outside the `String` statement. A `ParaLine` statement usually consists of `String` statements that contain text interspersed with statements for table anchors, frame anchors, markers, and cross-references.

About ID numbers

The table ID used by the `ATbl` statement must exactly match the ID given by the `TblID` statement. If it does not, the MIF interpreter ignores the `ATbl` statement and the table instance does not appear in the document. You cannot use multiple `ATbl` statements that refer to the same table ID.

An ID can be any positive integer from 1 to 65535, inclusive. The only other statements that require an ID are `AFrame` statements, linked `TextRect` statements, and `Group` statements. For more information about these statements, see "Graphic objects and graphic frames" on page 111.

Rotated cells

A table can have rotated cells and straddle cells. The following table includes rotated cells in the heading row:

Coffee	Price
Brazil Santos	$455.00

In a MIF file, a cell that is rotated simply includes a `CellAngle` statement that specifies the angle of rotation:

```
<Cell
    <CellAngle 270>
    <CellContent…>
>                                       # end of Cell
```

Cells can only be rotated by 90, 180, or 270 degrees. Cells are rotated clockwise.

Straddle cells

The contents of a straddle cell cross cell borders as if there were a single cell. You can straddle cells horizontally or vertically. The following table includes a heading row that straddles two columns:

Brazilian Coffee	
Coffee	**Price per Bag**
Brazil Santos	$455.00

The MIF code for the straddle cell includes a `CellColumns` statement that specifies the number of columns that the cell crosses. The contents of the straddle cell appear in the first of the straddle columns; the subsequent `Cell` statements for the row must appear even if they are empty.

```
<Row
    <Cell
        <CellColumns  2>               # Number of straddle columns.
        <CellContent                   # Content is in the first cell.
            <Para
                <PgfTag `CellHeading'>
                <ParaLine
                    <String `Brazilian Coffee'>
                >
            >                          # end of Para
        >                              # end of CellContent
    >                                  # end of Cell
    <Cell                              # Second cell appears, even though
        <CellContent                   # it is empty.
            <Para
                <PgfTag `CellHeading'>
                <ParaLine>
            >                          # end of Para
        >                              # end of CellContent
    >                                  # end of Cell
>                                      # end of Row
```

If the cell straddles rows, the substatement is `CellRows`.

Creating a table format

A table format includes the following properties:

- The properties specified by the Table Designer
- These include the row and column ruling and shading styles, the position of text within cell margins, the table's placement within the text column, and the table title position.
- The number and widths of columns
- The paragraph format of the first paragraph in the title (if there is one)
- The paragraph format of the topmost paragraph in the heading, body, and footing cell of each column

For example, you could change the format of the previous table to include shaded rows and a different ruling style:

Coffee	Price per Bag
Brazil Santos	$455.00
Celebes Kalossi	$924.00

Coffee	Price per Bag
Colombian	$474.35

The following MIF statements define this table format:

```
<TblFormat
    <TblTag `Coffee Table'>
                                        # Every table must have at least one TblColumn
# statement.
    <TblColumn
        <TblColumnNum  0>               # Columns are numbered from 0.
        <TblColumnWidth  2.0">          # Width of first column.
        >                               # end of TblColumn
    <TblColumn
        <TblColumnNum  1>               # Second column.
        <TblColumnWidth  1.5">          # Width of second column.
    >                                   # end of TblColumn
    <TblCellMargins  6.0 pt 6.0 pt 6.0 pt 4.0 pt>
    <TblLIndent  0.0">                  # These are exactly like paragraph
    <TblRIndent  0.0">                  # format properties.
    <TblAlignment Center >
    <TblPlacement Anywhere >
    <TblSpBefore  12.0 pt>
    <TblSpAfter  12.0 pt>
    <TblBlockSize  1>
    <TblHFFill  15>                     # No fill for heading row.
    <TblHFColor `Black'>
    <TblBodyFill  5>                    # Use 10% gray fill for main body rows.
    <TblBodyColor `Black'>
    <TblShadeByColumn No >              # Shade by row, not by column.
    <TblShadePeriod  1>                 # Shade every other row.
    <TblXFill  15>                      # No fill for alternate rows.
    <TblXColor `Black'>                 # Color for alternate rows.
    <TblAltShadePeriod  1>
    <TblLRuling `Thin'>                 # Use thin left outside rule.
    <TblBRuling `Thin'>                 # Use thin bottom outside rule.
    <TblRRuling `Thin'>                 # Use thin right outside rule.
    <TblTRuling `Medium'>               # Use medium top outside rule.
    <TblColumnRuling `Thin'>            # Use thin rules between columns.
    <TblXColumnRuling `Thin'>
    <TblBodyRowRuling `Thin'>           # Use thin rules between rows.
    <TblXRowRuling `Thin'>
    <TblHFRowRuling `'>                 # No rules between heading rows.
    <TblSeparatorRuling `Medium'>       # Use medium rule after heading row.
    <TblXColumnNum  1>
    <TblRulingPeriod  4>
    <TblLastBRuling No >
    <TblTitlePlacement InHeader>        # Place title above table.
    <TblTitlePgf1                       # Paragraph format for first
        <PgfTag `TableTitle'>           # paragraph in title.
    >                                   # end of TblTitlePgf1
    <TblTitleGap  6.0 pt>               # Gap between title and table.
    <TblInitNumColumns  2>              # Initial number of rows and
    <TblInitNumHRows  1>                # columns for new tables with
    <TblInitNumBodyRows  4>             # this format.
    <TblInitNumFRows  0>
    <TblNumByColumn No >
>                                       # end of TblFormat
```

The `TblColumn` statement numbers each column and sets its width. A table can have more columns than `TblColumn` statements; if a column does not have a specified format, the MIF interpreter uses the format of the most recently defined column.

Note: *A table instance must have at least one* `TblColumn` *statement. A table can use a format from the Table Catalog that includes a* `TblColumn` *statement or it can include a local* `TblFormat` *statement that supplies the* `TblColumn` *statement.*

Adding a Table Catalog

You can store table formats in a Table Catalog by using a `TblCatalog` statement. A document can have only one `TblCatalog` statement, which must occur before the `Tbls` statement.

The `TblCatalog` statement contains one `TblFormat` statement for each format, as shown in the following template:

```
<TblCatalog
    <TblFormat…>
<TblFormat…>
>       # end of TblCatalog
```

As with the Paragraph Catalog, if your MIF file does not provide a Table Catalog, the MIF interpreter supplies a default catalog and formats. If you do provide a Table Catalog, your defined table formats supersede those in the default Table Catalog.

You can add a minimal table format to the catalog by simply supplying a table format tag name. The MIF interpreter supplies a set of default values to the table's properties when it reads in the MIF file.

The ruling styles in a table format are defined in a separate catalog called the Ruling Catalog. You can define your own Ruling Catalog with the `RulingCatalog` statement. Whether you use the default ruling styles or create your own, substatements that refer to ruling styles, such as the `TblLRuling` statement, must use the name of a ruling style from the Ruling Catalog. See "RulingCatalog statement" on page 83.

Applying a table format

You can apply a table format from the Table Catalog or you can define a table format locally.

To apply a table format from the Table Catalog, use the `TblTag` statement within the `Tbl` statement:

```
<Tbls
    <Tbl
        <TblID 1>
        <TblTag `Format A'>             # Tag of format in Table Catalog
        <TblNumColumns 1>
        <TblBody
        ...
        >                               # end of TblBody
    >                                   # end of Tbl
>                                       # end of Tbls
```

To locally define a table format, use a complete `TblFormat` statement:

```
<Tbls
 <Tbl
  <TblID 1>
  <TblFormat
    <TblTag ` '>
                                        # Every table must have one TblColumn statement.
    <TblColumn
      <TblColumnNum 0>
      <TblColumnWidth  1.0">
    >                                   # end of TblColumn
    …table property statements…
```

```
  >                                    # end of TblFormat
 >                                     # end of Tbl
>                                      # end of Tbls
```

Creating default paragraph formats for new tables

You can use the `TblFormat` and `TblColumn` statements to define default paragraph formats for the columns in new tables. These default formats do not affect tables that are defined within the MIF file; they only affect tables that the user inserts after the MIF file has been opened in FrameMaker. Your filter or application should provide these defaults only for documents that might be edited later.

For example, the following MIF code assigns a paragraph format named Description to body cells in new tables that are given the format called Coffee Table:

```
<TblFormat
    <TblTag `Coffee Table'>
    <TblColumn
        <TblColumnNum 0>
        <TblColumnWidth  1.0">
        <TblColumnBody
            <PgfTag `Description'>
        >                          # end of TblColumnBody
    >                              # end of TblColumn
>                                  # end of TblFormat
```

Tables inherit properties differently

Tables inherit formatting properties somewhat differently than other document components. A table without an applied table format does not inherit one from a previously defined table. Instead, it gets a set of default properties from the MIF interpreter. Thus, if you apply a named format to a table, a following table will not inherit that format.

Paragraphs in table cells still inherit properties from previously defined paragraph formats. If you give a table cell a certain paragraph style, all subsequent cells inherit the same property unless it is explicitly reset. Table cells can inherit paragraph properties from any previously specified paragraph format, including other tables, paragraphs, or even the Paragraph Format catalog.

Tips

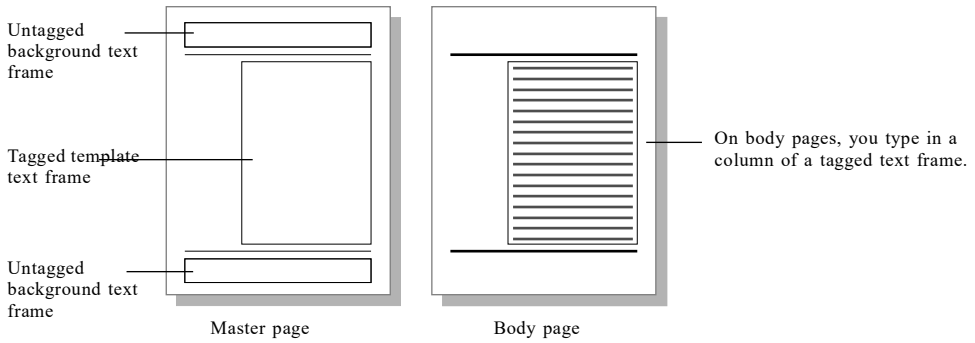
To avoid problems when creating tables:

- Give each table a unique ID number.
- Make sure that each `Tbl` statement has only one corresponding `ATbl` statement, and that each `ATbl` statement has a corresponding `Tbl` statement.
- Make sure that each `ATbl` statement matches the ID of its corresponding table instance.

Specifying page layout

FrameMaker documents have two kinds of pages that determine the position and appearance of text in the document: body pages and master pages.

Body pages contain the text and graphics that form the content of the document. Master pages control the layout of body pages. Each body page is associated with one master page, which specifies the number, size, and placement of the page's text frames and the page background, such as headers, footers, and graphics.



Text frames define the layout of the document's text on a page. A text frame can arrange text in one or more columns. In MIF, a text frame is represented by a `TextRect` statement. The dimensions of the text frame and the number of columns in the text frame are specified by substatements under the `TextRect` statement.

A text flow describes the text contained in one or more text frames. In MIF, a text flow is represented by a `TextFlow` statement. The actual text of the document is specified by substatements under the `TextFlow` statement.

If the text flow has the autoconnect property (if the text flow uses the MIF statement `<TFAutoConnect Yes>`), the text flow runs through a series of text frames; when you fill up one text frame, text continues into the next text frame. Most documents have only one text flow, although you can create many separate flows.

FrameMaker provides a default right master page for single-sided documents and default right and left master pages for double-sided documents. A MIF file can either use the default page layout or provide a custom layout.

Using the default layout

If you don't need to control the page layout of a document, you can use the default page layout by putting all of the document's text into a `TextFlow` statement. When reading the file, the MIF interpreter creates default master pages and body pages. The MIF file creates a single-column text frame for the body pages to contain the document's text. The MIF interpreter associates the text flow with this text frame.

The following example is in the sample file `defpage.mif`:

```
<MIFFile 2015>                          # Hand generated
<TextFlow                  # All document text is in this text flow.
    <TFTag `A'>               # Make this a tagged text flow.
    <TFAutoConnect Yes>       # Automatically connect text frames.
    <Para
        <ParaLine
            <String `This paragraph appears on a body page within a'>
            <String ` text flow tagged A.'>
        >                                 # end of ParaLine
    >                                     # end of Para
>                                         # end of TextFlow
                                          # End of MIFFile
```

A text flow must be tagged, and it must include `<TFAutoConnect Yes>`; otherwise, when the user adds text to the document, FrameMaker won't create additional pages and text frames to hold the added text.

Creating a simple page layout

If you want some control of the page layout but do not want to create master pages, you can use the `Document` substatements `DPageSize`, `DMargins`, and `DColumns` to specify the page size, margins, and number of columns in the text frame in the document. The MIF interpreter uses this information to create master pages and body pages. These statements correspond to the Normal Page Layout options.

The following example is in the sample file `columlay.mif`:

```
<MIFFile 2015>                          # Hand generated
<Document
    <DPageSize 7.5" 9.0">               # Set the page size.
    <DMargins 2" 1" .5" .5">            # Set the margins.
    <DColumns 1>                        # Set the number of columns in the default
                                        # text frame.
    <DTwoSides No>                      # Set document to single-sided.
>                                       # end of Document
<TextFlow                               # Document text is in this text flow.
    <TFTag `A'>                         # Make this a tagged text flow.
    <TFAutoConnect Yes>                 # Automatically connect text frames.
    <Para
        <ParaLine
            <String `This paragraph appears on a body page within a'>
            <String ` text flow tagged A.'>
        >                               # end of ParaLine
    >                                   # end of Para
>                                       # end of TextFlow
                                        # End of MIFFile
```

Creating a single-sided custom layout

If the document that you're importing needs a custom master page, you must specify a custom page layout. For example, a document might need a master page for background graphics.

To create a custom layout for a single-sided document, you do the following:

- Create a right master page.
- Create a single, empty body page.
- Create an empty, tagged text flow that is linked to the master page.
- Create a tagged text flow that is linked to the body page and contains all the document's text.

The MIF code shown in this section is also in the sample file `snglpage.mif`.

To create the master page

To create a master page layout, use the `Page` statement to create the page and use the `TextRect` statement to create the text frame.

To specify the number of text columns in the text frame, use the `TRNumColumns` statement. By default, if the text frame's specification does not include this statement, the text frame has only one column.

This example sets up a right master page with a text frame containing one text column:

```
<MIFFile 2015>                          # Hand generated
<Document
 <DPageSize 7.5" 9.0">                  # Set the document page size.
 <DTwoSides No>                         # Make this a single-sided document.
>                                       # end of Document
<Page                                   # Create a right master page.
 <PageType RightMasterPage>
 <PageTag `Right'>
```

```
 <TextRect                                  # Set up a text frame.
   <ID 1>                                   # Give the text frame a unique ID.
   <Pen 15>                                 # Set the pen style.
   <Fill 15>                                # Set the fill pattern (none).
   <ShapeRect 2" 1" 5" 7.5">                # Specify the text frame size.
   <TRNumColumns 1>                         # Specify number of text columns.
   <TRColumnGap 0.0">                       # Specify gap between text columns.
 >                                          # end of TextRect
>                                           # end of Page
```

The `ID` statement assigns a unique ID number to this text frame. You must give text frames a unique ID in a MIF file; other objects that require unique IDs are anchored graphic frames and table instances.

To create an empty body page

To create the body page, use the `Page` statement. Then use the `TextRect` statement to create a text frame with dimensions that are exactly the same as the text frame on the master page. Give the text frame a unique ID:

```
<Page
 <PageType BodyPage>
 <PageBackground `Default'>
 <TextRect
   <ID 2>                                   # This text frame has a unique ID.
                                            # The body page dimensions match those of the
                                            # master page.
   <ShapeRect 2" 1" 5" 7.5">
   <TRNumColumns 1>                         # The column layout must also match.
   <TRColumnGap 0.0">
 >                                          # end TextRect
>                                           # end Page
```

If the dimensions (specified by the `ShapeRect` statement) and column layout (specified by the `TRNumColumns` and `TRColumnGap` statements) of the master page and body page do not match, the body page will not use the page layout from the master page. Instead, the body page will use the page layout defined for the body page.

To create the text flow for the master page

The text flow for the master page is not contained in the `Page` statement; instead, it is contained in a `TextFlow` statement that is linked to the text frame on the master page. The `Page` statements must come before any `TextFlow` statements.

Link the text flow to the master page's text frame by using the `TextRectID` statement to refer to the text frame's unique ID:

```
<TextFlow
    <TFTag `A'>                         # The text flow must be tagged.
    <TFAutoConnect Yes>                 # Autoconnect must be turned on.
    <Para
        <ParaLine
            <TextRectID 1>              # Refers to text frame ID on master page.
        >                               # end of ParaLine
    >                                   # end of Para
>                                       # end of TextFlow
```

The text flow for the master page must be empty. Be sure to give the text flow the same flow tag that you give the text flow for the body page and to turn on the autoconnect feature.

To create the text flow for the body page

The text flow for the body page is contained in a separate `TextFlow` statement that is linked to the body page's text frame. The text flow contains the actual text of the document in one or more `Para` statements. If text overflows the first text frame, the MIF interpreter creates another body page with a layout that matches the right master page and pours text into the body page's text frame.

```
<TextFlow
 <TFTag `A'>
 <TFAutoConnect Yes>
 <Para
  <TextRectID 2>
  <PgfTag `Body'>
  <ParaLine
   <String `This appears on a body page within a text flow'>
   <String ` tagged A.'>
  >                              # end of ParaLine
 >                               # end of Para
>                                # end of TextFlow
```

Why one body page?

The method you use to create body pages is different from the method that FrameMaker uses when it writes a MIF file. When FrameMaker writes a file, it knows where each page break occurs in the file, so it creates a series of `Page` statements that each contain the text and graphics located on that page. When you are importing a document, you do not know where page breaks will fall, so you cannot break the document into a series of `Page` statements. Instead, you simply create one text flow for the entire document and link it to a single, empty body page. When the MIF interpreter reads the file, it creates as many pages as the document requires and gives each page the background specified by the master page.

Creating a double-sided custom layout

If you import a two-sided document, you might need to specify different page layouts for right and left pages. For example, a document might have a wider inside margin to allow extra room for binding. You can do this in a MIF file by creating and linking a second master page and a second body page. As with a single-sided layout, all the document's text is in one text flow. When the MIF interpreter reads the file, it adds alternate left and right body pages to the document. You can control whether the document starts with a right page or a left page by using the `DParity` statement.

For an example of a document with left and right master pages, see the sample file `dblpage.mif`.

Creating a first master page

In addition to left and right master pages, you can create custom master page layouts that you can apply to body pages. For example, some books have a special layout for the first page in a chapter.

In a MIF file, you can create as many master pages as you need, but you cannot apply all of them to the appropriate body pages. You can only apply a left page, a right page, and one additional custom master page to the body pages. Furthermore, you can only link the custom master page to the first page in a document.

When you are importing a document into FrameMaker, you do not know how much text the MIF interpreter will put on a page; you can only determine where the first page begins. When the interpreter reads the MIF file, it applies the custom master page layout to the first page in the document. For each subsequent page, it uses the `DParity` and `DTwoSides` statements to determine when to add a left page and when to add a right page.

Other master page layouts that you've defined are not lost when the interpreter reads a MIF file. The user can still apply these page layouts to individual body pages.

For an example of a MIF file with a first page layout, see the sample file `frstpage.mif`.

Adding headers and footers

Headers and footers are defined in untagged text flows on the master pages of a document. When FrameMaker creates default master pages, it automatically provides untagged text flows for headers and footers.

If you are importing a document that has headers and footers, you define additional text frames on the master pages. Link an untagged text flow to each additional text frame on the master page. The untagged text flow contains the text of the header or footer.

For an example of a MIF file with a footer, see the sample file `footers.mif`. Note that the footer text flow contains a variable; you can place variables only in untagged text flows on a master page, not in tagged flows.

Creating markers

A FrameMaker document can contain markers that hold hidden text and mark locations. For example, you use markers to add index entries, cross-references, and hypertext commands to a document. FrameMaker provides both predefined marker types and markers that you can define as needed. (For more information about markers and marker types, see page 136.)

Within a FrameMaker document, you insert a marker by choosing the Marker command from the Special menu. In a MIF file you insert a marker by using a `Marker` statement. The `Marker` statement specifies the marker type and the marker text.

The following example inserts an index marker:

```
<Para
    <ParaLine
        <Marker
            <MType 2>                   # Index marker
            <MText `Hello world'># Index entry
        >                               # end of Marker
        <String `Hello world'>
    >                                   # end of ParaLine
>                                       # end of Para
```

The `MText` statement contains the complete index entry.

When FrameMaker writes a `Marker` statement, the statement includes an `MCurrPage` substatement with the page number on which the marker appears. You do not need to provide an `MCurrPage` statement when you generate a MIF file; this statement is ignored when the MIF interpreter reads a MIF file.

Creating cross-references

In a FrameMaker document, you can create cross-references that are automatically updated. A cross-reference can refer to an entire paragraph or to a particular word or phrase in a paragraph. The text to which a cross-reference points is called the *reference source*; the actual location of the cross-reference is the *reference point*.

The format of a cross-reference determines its appearance and the wording. Cross-reference formats include *building blocks,* instructions to FrameMaker about what information to extract from the reference source. A common building block is `<$pagenum>`, which FrameMaker replaces with the page number of the reference source. Another common building block is <$paratext>, which FrameMaker replaces with the text content of the paragraph, excluding autonumbering and special characters such as tabs and forced line breaks.

Within a FrameMaker document, you insert and format cross-references by choosing Cross-Reference from the Special menu. In a MIF file, you create a cross-reference as follows:

- Create the format of cross-references by using `XRefFormats` and `XRefFormat` statements.
- Insert a marker at the reference source by using a `Marker` statement.
- Insert the reference point by using an `XRef` statement.

Creating cross-reference formats

The cross-reference formats for a document are defined in one `XRefFormats` statement. A document can have only one `XRefFormats` statement.

The `XRefFormats` statement contains one or more `XRefFormat` statements that define the cross-reference formats. A cross-reference format consists of a name and a definition.

```
<XRefFormats
    <XRefFormat
        <XRefName `Page'>
        <XRefDef `page\x11 <$pagenum\>'>
    >                           # end of XRefFormat
>                               # end of XRefFormats
```

The name can be any string allowed in a MIF file (see "Character set in strings" on page 7). In this example, a nonbreaking space (`\x11`) appears between the word "page" and the page number. Each cross-reference format must have a unique name; names are case-sensitive. The cross-reference definition contains text and cross-reference building blocks. See your user's manual or the online Help system for a list of building blocks.

Inserting the reference source marker

To mark the location of the reference source, insert a `Marker` statement at the beginning of the reference source. The following example creates a cross-reference to a heading:

```
<Para
    <PgfTag `Heading'>
    <ParaLine
        <Marker
            <MType 9>               # Identifies this as a cross-reference
            <MText `34126: Heading: My Heading'>
                                    # Cross-reference source
        >                           # end of Marker
        <String `My Heading'>
    >                               # end of ParaLine
>                                   # end of Para
```

The `<MType 9>` statement identifies this as a cross-reference marker; it is required. The `MText` statement contains the cross-reference source text, which must be unique. When FrameMaker writes a cross-reference, it adds a unique number and the paragraph tag to the `MText` statement, as shown in the previous example. While the number is not required, it guarantees that the cross-reference points to a unique source when the number is present. In the previous example, the number in `<MText>` is not mandatory. However, the number in the example ensures that the new cross-reference points to the `'My heading'` heading.

Inserting the reference point

The final step in creating a cross-reference is to insert an `XRef` statement at the position in text where the cross-reference should appear. The `XRef` statement provides the name of the cross-reference format (defined in `XRefFormat`), the source text, and the pathname of the file containing the source:

```
<Para
    <PgfTag `Body'>
    <ParaLine
        <String `This is a cross-reference to '>
        <XRef
            <XRefName `Page'>          # Cross-reference format
            <XRefSrcText `34126: Heading: My Heading'>
                                       # Source text
            <XRefSrcFile `'>           # File containing source
        >                              # end of XRef
        <XRefEnd>
        <String `.'>
    >                                  # end of ParaLine
>                                      # end of Para
```

The format name must exactly match the name of a format defined in `XRefFormats`. The source text must be unique and must match the string in the `MText` statement in the corresponding reference point marker. The `XRefSrcFile` statement is only required if the reference source is in a different file from the reference point. It must be a valid MIF filename (see "Device-independent pathnames" on page 7).

You must also supply an `XRefEnd` statement after the `XRef` statement.

How FrameMaker writes cross-references

When FrameMaker writes a cross-reference, it provides the actual text that will appear at the reference point. This information is not required in a MIF input file. The previous example would be written as follows:

```
<XRef
    <XRefName `Page'>
    <XRefSrcText `34126: Heading: My Heading'>
    <XRefSrcFile `'>
    >                              # end of XRef
<String `page'>                    # The text that appears in the document;
<Char HardSpace >                  # in this case, a page number followed a
<String `1'>                       # hard space and the number 1
<XRefEnd>                          # End of cross-reference text
```

If you do include the text of the cross-reference, make sure that the `XRefEnd` statement follows the text. FrameMaker considers everything between the `XRef` statement and the `XRefEnd` statement to be part of the cross-reference.

Creating variables

In a FrameMaker document, variables act as placeholders for text that might change. For example, many documents use a variable for the current date. A variable consists of a name, which is how you choose a variable, and a definition, which contains the text and formatting that appear where a variable is inserted.

FrameMaker provides two kinds of variables: system variables that are predefined by FrameMaker, and user variables that are defined by the user. System variables contain building blocks that allow FrameMaker to extract certain information from the document or the system, such as the current date or the current page number, and place it in text. Headers and footers frequently use system variables. You can modify a system variable's definition but you cannot create new system variables. User variables contain only text and formatting information.

Within a FrameMaker document, you insert and define variables by choosing Variable from the Special menu. The variable appears in the document text where it is inserted.

In a MIF file, you define and insert variables as follows:

- Define and name the document variables by using `VariableFormats` and `VariableFormat` statements.
- Insert the variable in text by using the `Variable` statement.

Defining user variables

All variable definitions for a document are contained in a single `VariableFormats` statement. The `VariableFormats` statement contains a `VariableFormat` statement for each document variable. The `VariableFormat` statement provides the variable name and definition.

```
<VariableFormats
    <VariableFormat
    <VariableName `Product Number'>
    <VariableDef `A15-24'>
    >               # end of VariableFormat
>                   # end of VariableFormats
```

The variable name must be unique; case and spaces are significant. For a user variable, the variable definition can contain only text and character formats; you can provide any character format defined in the Character Catalog. The following example applies the default character format Emphasis to a variable:

```
<VariableFormat
    <VariableName `Product Number'>
    <VariableDef `<Emphasis\>A15-24<Default ¶ Font\>'>
>               # end of VariableFormat
```

You can specify character formats as building blocks; that is, the character format name must be enclosed in angle brackets. Because of MIF parsing requirements, you must use a backslash sequence for the closing angle bracket.

Using system variables

Whenever you open or import a MIF file, the MIF interpreter provides the default system variables. You can redefine a system variable but you cannot provide new system variables.

System variables are defined by a `VariableFormat` statement. For example, the following statement shows the default definition for the system variable Page Count:

```
<VariableFormat
    <VariableName `Page Count'>
    <VariableDef `<$lastpagenum\>'>
>               # end of VariableFormat
```

System variables contain building blocks that provide certain information to FrameMaker. These building blocks are preceded by a dollar sign (`$`) and can only appear in system variables. Some system variables have restrictions on which building blocks they can contain. These restrictions are discussed in your user's manual and in the online Help system. You can add any text and character formatting to any system variable.

Inserting variables

To insert a user variable or a system variable in text, use the `Variable` statement. The following example inserts the system variable Page Count into a paragraph:

```
<Para
    <ParaLine
        <String `This document has '>
        <Variable
            <VariableName `Page Count'>
```

```
        >                   # end of Variable
        <String `pages.'>
    >                       # end of ParaLine
>                           # end of Para
```

The `VariableName` string must match the name of a variable format defined in the `VariableFormats` statement.

Variables are subject to the following restrictions:

- You cannot place any variable in a tagged text flow on a master page.
- The system variable `Current Page #` and the system variables for running headers and footers can only appear in untagged text flows on a master page.
- The system variables Table Continuation and Table Sheet can only appear in tables.

Creating conditional text

You can produce several slightly different versions of a document from a single conditional document. In a conditional document, you use condition tags to differentiate conditional text (text that is specific to one version of the document) from unconditional text (text that is common to all versions of the document).

In a MIF file, you create a conditional document as follows:

- Create the condition tags to be used in the document and specify their format via `ConditionCatalog` and `Condition` statements.
- Apply one or more condition tags to the appropriate sections of the document via `Conditional` and `Unconditional` statements.
- Show or hide conditional text by using the `CState` statement.

Creating and applying condition tags

In MIF, all condition tags are defined in a `ConditionCatalog` statement, which contains one or more `Condition` statements. A `Condition` statement specifies the condition tag name, the condition indicators (how conditional text appears in the document window), a color, and a state (either hidden or shown).

For example, the following statements create a Condition Catalog with two conditional tags named Summer and Winter:

```
<ConditionCatalog
    <Condition
        <CTag `Summer'>                 # Condition tag name
        <CState CHidden >               # Condition state (now hidden)
        <CStyle COverline >             # Condition indicator
        <CColor `Blue'>                 # Condition indicator
    >                                   # end of Condition
    <Condition
        <CTag `Winter'>
        <CState CShown >                # This condition is shown
        <CStyle CUnderline >
        <CColor `Red'>
    >                                   # end of Condition
>                                       # end of ConditionCatalog
```

To mark conditional and unconditional passages within document text, use `Conditional` and `UnConditional` statements as shown in the following example:

```
<Para
    <ParaLine
        <String `Our company makes a full line of '>
```

```
                                        # Unconditional text
        <Conditional                    # Begin conditional text
            <InCondition `Winter'>      # Specifies condition tag
        >                               # end of Conditional
        <String `warm and soft sweaters'>
                                        # Conditional text
        <Conditional                    # Begin conditional text
            <InCondition `Summer'>      # Specifies condition tag
        >                               # end of Conditional
        <String `cool and comfortable tank tops'>
        <Unconditional >
        <String ` for those '>          # Unconditional text
    >                                   # end of ParaLine
    <ParaLine
        <Conditional
            <InCondition `Winter'>
        >                               # end of Conditional
        <String `chilly winter'>
        <Conditional
            <InCondition `Summer'>
        >                               # end of Conditional
        <String `hot summer'>
        <Unconditional >
        <String ` days.'>
    >                                   # end of ParaLine
>                                       # end of Para
```

You can apply multiple condition tags to text by using multiple `InCondition` statements:

```
<Conditional
    <InCondition `Winter'>
    <InCondition `Summer'>
>               # end of Conditional
```

Showing and hiding conditional text using Boolean expressions

You can also use Boolean expressions to show or hide conditional text. Boolean condition expressions are identified using the `BoolCondTag`. You can create these expressions by linking condition tags with boolean operators and describe them in the `BoolCondExpr` statement. If the value of `BoolCondState` of a Boolean condition expression is set to 'Active' the show/hide state of the text in that document is governed by that Boolean condition expression. All text for which the expression evaluates to 'True' is shown, while the rest are hidden.

Consider a scenario where you have created Conditions summary, detail, comment, and a boolean expression "comment"OR"summary"OR"detail". If the value of `BoolCondState` is 'Active', FrameMaker uses this expression to determine the Show/Hide state of conditional text.

The `BoolCond` statement appears in the `BoolCondCatalog` as shown below :

```
<BoolCond
<BoolCondTag `Conditional Expression'>
<BoolCondExpr `"comment"OR"summary"OR"detail"'>
<BoolCondState `Active'>
```

> # end of BoolCond

When you save a FrameMaker 8 document as MIF, the following system tags are displayed in the MIF:

- FM8_SYSTEM_HIDEELEMENT
- FM8_TRACK_CHANGES_ADDED
- FM8_TRACK_CHANGES_DELETED

Note: *These tags are used by the system and are reserved for internal use only.*

How FrameMaker writes a conditional document

If you are converting a MIF file that was generated by FrameMaker, you need to understand how FrameMaker writes a file that contains hidden conditional text.

When FrameMaker writes a MIF file, it places all hidden conditional text in a text flow with the tag name `HIDDEN`. Within the document text flow, a conditional text marker, `<Marker <MType 10>>`, indicates where hidden conditional text would appear if shown.

The marker text contains a plus sign (+) followed by a unique five-digit integer. The corresponding block of hidden text is in the hidden text flow. It begins with a conditional text marker containing a minus sign (–) and a matching integer and ends with a marker containing an equal sign (=) and the same integer. One or more `Para` statements appear between the markers. If the hidden conditional text doesn't span paragraphs, all the text appears in one `Para` statement. If the hidden text spans paragraphs, each end of paragraph in the conditional text forces a new `Para` statement in the hidden text flow.

The following example shows how FrameMaker writes the sentence used in the previous example:

```
                                # This text flow contains the sentence as it appears in
                                # the document body.
<TextFlow
    <TFTag `A'>
    <TFAutoConnect Yes >
    <Para
        <ParaLine
            <String `Our company makes a full line of '>
                                # This marker indicates that hidden text appears in the
                                # hidden text flow.
            <Marker
                <MType  10>
                <MText `+88793'>
                <MCurrPage  0>
            >                   # end of Marker
            <Conditional
                <InCondition `Summer'>
            >                   # end of Conditional
            <String `cool and comfortable tank tops'>
            <Unconditional >
            ...
        >                       # end of Para
>                               # end of TextFlow
                                # This text flow contains the hidden conditional text.
<TextFlow
    <TFTag `HIDDEN'>
    <Para
        <PgfEndCond Yes >
        <ParaLine
            <Marker
                <MType  10>
                                # This marker shows the beginning of hidden text.
                                # Its ID matches the marker ID in the body text flow.
                <MText `-88793'>
                <MCurrPage  0>
            >                   # end of Marker
            <Conditional
                <InCondition `Winter'>
            >                   # end of Conditional
                                # Here's the hidden text.
            <String `chilly winter'>
            <Marker
                <MType  10>
```

```
                                # This marker shows the end of hidden text. It must
                                # match the marker that begins with a minus sign (-).
                <MText `=88793'>
                <MCurrPage  0>
            >                   # end of Marker
        >
    >                           # end of Para
    ...
>                               # end of TextFlow
```

Creating filters

Structured FrameMaker allows specific components in a structured document to be processed differently to generate different output formats. Consider a case where you want some text in a document to be included in the Print output, but not in the HTML Help output. You can create a filter based on the values of the attributes of elements, and process only those elements in the document that match the filter, and include such elements in the Print output.

In a MIF file, you create a filter required for generating the output of a structured document using the `DefAttrValuesCatalog`, `DefAttrValues`, `AttrCondExprCatalog`, and `AttrCondExpr` statements.

All MIF 8 documents contain a catalog of predefined filters. The catalog is empty if a filter is not defined in a structured document. A filter comprises a tag called `AttrCondExprTag`, the expression tag `AttrCondExprStr`, and the state of the filter which is stored in the `AttrCondState` tag. The state of the filter indicates whether the filter is active in the document. Although the catalog can have several filters, only one filter must be active at any time.

To create filters, use the `AttrCondExprCatalog` statement as illustrated in the following example where two filters are created:

```
<AttrCondExprCatalog
 <AttrCondExpr
  <AttrCondExprTag `NewExpr1'>
  <AttrCondExprStr `(A="val1" OR A="val11") AND (B="val2" OR B="val22")'>
  <AttrCondState `Inactive'>
 > # end of AttrCondExpr
 <AttrCondExpr
  <AttrCondExprTag `NewExpr2'>
  <AttrCondExprStr `(A="val4" OR A="val44") OR (B="val3" OR B="val33")'>
  <AttrCondState `Active'>
 > # end of AttrCondExpr
> # end of AttrCondExprCatalog
```

The following statements create an empty filter catalog:

```
<AttrCondExprCatalog
```

> # end of AttrCondExprCatalog

All MIF 8 documents contain attribute-value pairs.

To create a catalog of attributes with values, use the `DefAttrValuesCatalog` statement as illustrated in the following example:

```
<DefAttrValuesCatalog
 <DefAttrValues
  <AttributeTag `A'>
  <AttributeValue `val1'>
  <AttributeValue `val2'>
 > # end of DefAttrValues
 <DefAttrValues
  <AttributeTag `B'>
  <AttributeValue `val3'>
```

```
  <AttributeValue `val4'>
 > # end of DefAttrValues
> # end of DefAttrValuesCatalog
```

The following statements create a catalog of attributes without values:

```
<DefAttrValuesCatalog
```

> # end of DefAttrValuesCatalog

Including template files

When you write an application, such as a filter or a database publishing application, to generate a MIF file, you have two ways to include all formatting information in the file:

- Generate all paragraph formats and other formatting information directly from the application.
- Create a template document in FrameMaker, save it as a MIF file, and include the template file in your generated MIF file.

It's usually easier to create a template in FrameMaker than it is to generate the formatting information directly.

To create the template as a MIF file, do the following:

1 Create the template in FrameMaker and save it as a MIF file.

2 Edit the MIF file to preserve the formatting catalogs and the page definitions and delete the text flow.

3 Generate the text flow for your document and use the `include` statement to read the formatting information from the template.

Creating the template

Create the template document in FrameMaker. Define the paragraph and character formats, table formats, variable and cross-reference formats, master pages, and any other formatting and page layout information that your document needs. Generally, a template contains some sample lines that illustrate each format in the document. Save the completed template as a MIF file. For more information about creating templates, see your user's manual.

Editing the MIF file

You need to edit the resulting MIF file to extract just the formatting and page layout information.

1 Delete the MIFFile statement.

2 Search for the first body page and locate its TextRect statement.

To find the first body page, search for the first occurrence of `<PageType BodyPage>`. Suppose the first body page in your MIF file looks like this:

```
<Page
 <Unique 45155>
 <PageType BodyPage >
 <PageNum `1'>
 <PageSize  8.5" 11.0">
 <PageOrientation Portrait >
 <PageAngle 0.0>
 <PageBackground `Default'>
 <TextRect
  <ID 7>
  <Unique 45158>
  <Pen 15>
  <Fill 15>
```

```
  <PenWidth  1.0 pt>
  <ObColor `Black'>
  <DashedPattern
   <DashedStyle Solid>
  >                                 # end of DashedPattern
  <ShapeRect  1.0" 1.0" 6.5" 9.0">
  <TRNext 0>
 >                                  # end of TextRect
>                                   # end of Page
```

The ID for the `TextRect` on this body page is 7. Remember this ID number. If there is more than one `TextRect` on the body page, remember the ID of the first one.

3 Locate the text flow associated with the TextRect statement on the first body page and delete it.

Suppose you are working with the previous example. You would search for the statement `<TextRectID 7>` to locate the text flow. It might look similar to the following:

```
<TextFlow
    <Notes>                         # end of Notes
    <Para
        <Unique  45157>
        <PgfTag `MyFormat'>
        <ParaLine
            <TextRectID  7>
            <String `A single line of text.'>
        >
    >                               # end of Para
>                                   # end of TextFlow
```

Delete the entire text flow.

4 From your application, generate a MIF file that includes the edited template file.

Suppose the edited MIF file is called `mytemplate.mif`. Your application would generate the following two lines at the top of any new MIF file:

```
<MIFFile 2015>                      # Generated by my application
include (mytemplate.mif)
```

The `include` statement is similar to a C `#include` directive. It causes the MIF interpreter to read the contents of the file named `mytemplate.mif`. For more information about filenames in MIF, see "Device-independent pathnames" on page 7.

5 From your application, generate a text flow that contains the entire document contents.

The text flow should use the ID and tag name of the text flow you deleted from the template file; this associates the new text flow with the first body page in the template.

The entire generated MIF file would look something like this:

```
<MIFFile 2015>                      # Generated by my application
include (mytemplate.mif)
<TextFlow
    <TFTag `A'>
    <TFAutoConnect Yes>
    <TextRectID 7>
    <Para
        <ParaLine
            <String `This is the content of the generated document.'>
        >
    >                               # end of Para
>                                   # end of TextFlow
```

A user can open the generated MIF file to get a fully formatted FrameMaker document.

Setting View Only document options

You can use MIF statements to control the display of View Only documents. A View Only document is a locked FrameMaker hypertext document that a user can open, read, and print but not edit. You can use MIF statements to control the appearance and behavior of the document window and to control the behavior of cross-references in locked documents.

The MIF statements for View Only documents are intended for hypertext authors who want more control over hypertext documents. They do not have corresponding commands in the user interface.

The View Only MIF statements described in this section must appear in a `Document` statement. These statements have no effect in an unlocked document. Make sure that the `Document` statement also includes the following substatement:

```
<DViewOnly Yes>
```

Changing the document window

You can use MIF statements to change the appearance and behavior of the document window in the following ways:

- To suppress the document window menu bar, use the following statement:

```
<DViewOnlyWinMenubar No>
```

This statement has no effect in the Windows version of FrameMaker because those versions have an application menu bar rather than a document window menu bar.

- To suppress the display of scroll bars and border buttons in the document window, use the following statement:

```
<DViewOnlyWinBorders No>
```

- To suppress selection in the document window, include the following statement:

```
<DViewOnlySelect No>
```

You can normally select text and objects in a locked document by Control-dragging in UNIX and Windows versions. Specifying `<DViewOnlySelect No>` prevents all selection in a locked document.

- To suppress the appearance of a document region pop-up menu, use the statement:

```
<DViewOnlyWinPopup No>
```

A *document region pop-up menu* is a menu activated by the right mouse button. For example, in UNIX versions of FrameMaker, the Maker menu can be accessed by pressing the right mouse button. If the `DViewOnlyWinPopup` statement has a value of `No`, the background menu does not appear when the right mouse button is pressed. This statement has no effect in the Windows version of FrameMaker.

- To make a window behave as a palette window, use the following statement:

```
<DViewOnlyWinPalette Yes>
```

A *palette window* is a command window, such as the Equations palette, that exhibits special platform-dependent behavior. In UNIX versions of FrameMaker, a palette window can only be dismissed; it cannot be closed to an icon.

In Windows versions, a palette floats outside the main application window and cannot be unlocked. To edit the palette, you need to reset the `DViewOnlyWinPalette` statement to `No` in the MIF file before opening it in FrameMaker.

Using active cross-references

A locked document automatically has active cross-references. An *active cross-reference* behaves like a hypertext `gotolink` command; when the user clicks on a cross-reference, FrameMaker displays the link's destination page. By default, the destination page is shown in the same document window as the link's source.

You can use MIF statements to turn off active cross-references and to change the type of hypertext link that the cross-reference emulates. (By default, cross-references emulate the `gotolink` behavior.)

- To make cross-references emulate the `openlink` command, which displays the destination page in a new document window, use the following statement:

```
<DViewOnlyXRef OpenBehavior>
```

Use this setting to allow users to see both the source page and the destination page.

- To turn off active cross-references, use the following statement:

```
<DViewOnlyXRef NotActive>
```

Use this setting to emulate the behavior in earlier FrameMaker versions.

You can use the `DViewOnlySelect` statement to control whether active cross-references highlight the marker associated with destination text.

- When cross-references are active and `<DViewOnlySelect Yes>` is specified, clicking a cross-reference in the document highlights the marker associated with the destination text.
- When cross-references are active and `<DViewOnlySelect UserOnly>` is specified, clicking a cross-reference does not highlight the marker. However, the user can select text in the locked document.
- When cross-references are active and `<DViewOnlySelect No>` is specified, clicking a cross-reference does not highlight the marker. The user cannot select text in the locked document.

By default, clicking a cross-reference does not highlight the marker associated with the destination text but the user can select text in the locked document.

Disabling commands

You can disable specific commands in a View Only document. For example, a hypertext author might disable copy and print commands for sensitive documents.

To disable a command, you must supply the hex code, called an *fcode*, that internally represents that command in FrameMaker. For example, you can disable printing, copying, and unlocking the document by supplying the following statements:

```
<DViewOnlyNoOp 0x313># Disable printing
<DViewOnlyNoOp 0x322># Disable copying
<DViewOnlyNoOp 0xF00># Disable unlocking the document
```

The following table lists the files where you can find fcodes for commands:

For this version	Look here
UNIX	`$FMHOME/fminit/`*`language`*`/configui/Commands`, where *`language`* is the language in use, such as `usenglish`
Windows	*install_dir*`/fminit/configui/cmds.cfg`, where *install_dir* is the directory where FrameMaker is installed

See the online manual *Customizing FrameMaker* for more information about the commands file in UNIX versions.

Applications of MIF

You can use MIF files any time you need access to FrameMaker's formatting capabilities. This section provides some examples of how MIF can be used and some tips on minimizing MIF statements.

You can use MIF to:

- Share files with earlier versions of FrameMaker
- Perform custom document processing
- Write import and export filters for FrameMaker documents
- Perform database publishing

Sharing files with earlier versions

FrameMaker automatically opens documents created with an earlier version of FrameMaker (2.0 or higher).

To use an earlier version of FrameMaker (such as 5.5) to edit a document created with a later version of FrameMaker (such as 7.0):

1 Use the newer FrameMaker product version to save the document in MIF.

2 Open the MIF file with the earlier version of FrameMaker.

Note: *Earlier versions of FrameMaker do not support all MIF statements in the current version. For example, when you use version 5.5.6 or earlier of FrameMaker to open a document created in version 6.0 or later, MIF statements specifying optimized PDF size are skipped. You can ignore the related error messages. However, to regain the optimized PDF size you will need to use the* ***Optimize Pdf Size*** *command. For a description of the differences between MIF 7.0 and previous versions, see , "MIF Compatibility."*

Modifying documents

You can use MIF to perform custom document processing. For example, you can create a program or write a series of text editor macros to search for and change paragraph tags in a MIF file. You can also edit a MIF book file to easily add or change document names in a book.

For an example of using MIF to easily update the values in a table, see "Updating several values in a table" on page 240.

Writing filters

MIF allows you to write filters to convert data from other formats to FrameMaker format and to convert a MIF file to another document format. While FrameMaker will change in future versions, MIF will always remain compatible with earlier versions, so your filters can continue to write MIF files.

Import filters

MIF statements can completely describe a FrameMaker document or book file. Because documents created with most word processors and text editors have fewer features than a FrameMaker document, your import filters normally use only a subset of MIF statements.

To write an import filter, first determine which MIF statements describe the format of the input file. Then write a program to translate the file from its original file format to MIF. If the imported document doesn't use sophisticated formatting and layout features, don't include the corresponding MIF statements in your filter.

For example, if the file was created by a word processor, your filter should convert document text to a single `TextFlow` statement. Ignore line and page breaks (except forced breaks) in your source document, because the text will be repaginated by the MIF interpreter. If the document uses style sheets, convert paragraph styles to paragraph formats in a `PgfCatalog` statement, and convert table styles to table formats in a `TblCatalog` statement.

Output filters

You can write output filters that convert a MIF file to any format you want. While you should be familiar with all MIF statements to determine which ones you need to translate a FrameMaker document, your output filter doesn't need to convert all the possible MIF statements.

In most cases, a MIF description of a FrameMaker document contains more information than you need. Because MIF appears as a series of nested statements, your output filter must be able to scan a MIF file for the information it needs and skip over statements that it will not use.

Installing a filter

In UNIX versions, you can set up FrameMaker to automatically start a script that runs a filter based on the filename suffix. The filter can convert a file to a MIF file. FrameMaker then interprets the MIF file, storing the results in a FrameMaker document. For more information about installing your filter, see the online manual *Customizing FrameMaker*.

Minimizing MIF statements

The following tips may help you minimize the number of MIF statements that your filter needs to generate:

- If you are not concerned about controlling the format of a document, use the default formats that FrameMaker provides for new documents. The user can always change formats as needed within the FrameMaker document.
- If you are filtering a document from another application into FrameMaker and then back to the application, you may want to import the filter's MIF file into a FrameMaker document, save the document as a MIF file, and then convert the file back to the original format from the MIF file generated by FrameMaker. This technique takes advantage of FrameMaker's syntactically complete MIF statements, but allows your filter to write a shorter MIF file.
- If your filter needs to generate fully-formatted MIF files, you can minimize the number of formatting statements by creating a template in FrameMaker, saving the template as a MIF file, and then including the MIF template file in your filter's generated document. You must edit the saved MIF template (see "Including template files" on page 45). An advantage of this technique is that you can use the same template for more than one document.
- Define macros to ease the process of generating statements. For an example of using macros, see "Text example" on page 231.

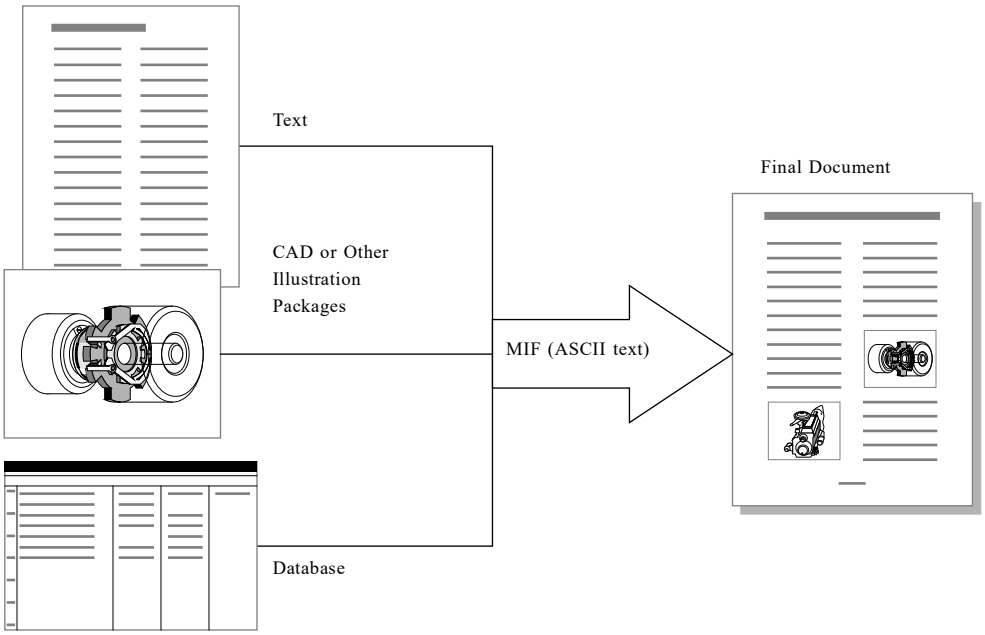
Database publishing

You can use MIF files to import information from an external application, such as a database, into a FrameMaker document. This type of information transfer is often called *database publishing*. For example, you can write a C program or a database script to retrieve information from a database and store that information as a MIF file. A user can then open or import the MIF file to get a fully formatted FrameMaker document that contains up-to-date information from the database.

There are four key elements to a typical database publishing solution:

- The database provides a system to enter, manipulate, select, and sort data. You can use any database that can create text-based output files.
- MIF provides the data interchange format between the database and FrameMaker. MIF can completely describe a document in ASCII format, including information such as text and graphics, page layout, and indexes and cross-references.
- FrameMaker provides the text formatting. FrameMaker reads MIF files and dynamically manages line breaks, page breaks, headers and footers, and graphics. The user can view, print, save, or even navigate through an online document using hypertext commands.

- Optional control programs allow you to tightly integrate the database and FrameMaker. Some database publishing applications are controlled entirely from the database system or through hypertext commands embedded in a FrameMaker document. More complicated applications may require an external control program, such as a C program that issues queries and selects a FrameMaker document template.



For an example of a database publishing application, see "Database publishing" on page 241.

Debugging MIF files

When FrameMaker reads a MIF file, it might detect errors such as unexpected character sequences. In UNIX and Windows versions, FrameMaker displays messages in a console window. In the Windows version, you must turn on Show File Translation Errors in the Preferences dialog box to display messages in a window. If FrameMaker finds an error, it continues to process the MIF file and reads as much of the document as possible.

When you are debugging MIF files, you should examine the error messages for clues. The MIF interpreter reports line numbers for most errors. For a description of MIF error messages, see , "MIF Messages."

In some cases, the MIF interpreter reports an "invalid opcode" message for a statement. If the statement seems correct to you, check the statements above it. A missing right angle bracket can cause the interpreter to parse a statement incorrectly.

If the MIF interpreter brings up an empty document when it reads your file, it has stopped trying to interpret your file and opened an empty custom document instead. Close the document and check your MIF file for errors. Try adding a `Verbose` statement to your file to get more complete messages.

If your MIF statements are syntactically correct but cause unexpected results in the document, check for mismatched ID numbers and check the placement of statements. Many MIF statements are position-dependent and can cause errors if they appear in the wrong place in a file. For example, an `ATbl` statement that comes before its corresponding `Tbl` statement causes an error.

Here are some additional tips for debugging MIF files:

- Use the `Verbose` statement to generate comments. To debug a specific section of a MIF file, you can precede the section with the `<Verbose Yes>` statement and end the section with the `<Verbose No>` statement.
- Make sure angle brackets are balanced.
- Make sure that MIF statement names are capitalized correctly. MIF statement names and keyword values are case-sensitive.
- Make sure that string arguments are enclosed in straight single quotation marks. (See "MIF data items" on page 5 for an example.)
- Make sure ID numbers are unique.
- Make sure that every table anchor has a corresponding table instance, and that every table instance has an anchor in the text flow.
- Make sure that tag names with spaces are enclosed in straight single quotation marks.
- Make sure paired statements are balanced. For example, `XRef` and `XRefEnd` statements must be paired.
- Make sure that right angle bracket (>) and backslash (\) characters in text are preceded by a backslash.
- Make sure that hexadecimal characters, for example `\xe6`, have a space after them.

Other application tools

The Frame Developer's Kit (FDK) provides tools that you can use to write filters and to perform custom document processing. The FDK includes the Application Program Interface (API), which you can use to create a C application that can create and save documents, modify documents, and interact with the user. The FDK also includes the Frame Development Environment (FDE), which allows you to make your FDK clients portable to the platforms that FrameMaker supports.

MIF files can be used by C applications, text processing utilities, or UNIX shell scripts. You might want to work directly with MIF files if you are filtering large numbers of files in batch mode. You also might want to work with MIF files if you are doing simple document processing, such as changing a few tag names, or if you are setting options for View Only documents.

You can use the FDK and MIF files together; for example, a database publishing application can extract values from a database and write out the information as a table in a MIF file. An FDK client can then automatically open the MIF file as a FrameMaker document.

Where to go from here

This chapter has given you a start at working with MIF files. You can use the information in this chapter as guidelines for working with similar MIF statements. Once you have experimented with basic MIF files, you can learn about other MIF statements by creating small FrameMaker documents that contain a specific feature and saving these documents as MIF files. Because FrameMaker writes complete and precise MIF code, it is your ultimate source for learning about MIF statements.

For more information about document components not described in this chapter, see the MIF statement descriptions in , "MIF Document Statements", , "MIF Book File Statements", and , "MIF Statements for Structured Documents and Books".

Chapter 3: MIF Document Statements

This chapter describes the structure of MIF document files and the MIF statements they can contain. Most MIF statements are listed in the order that they appear in a MIF file, as described in the following section. If you are looking for information about a particular statement, use this manual's statement index to locate it. If you are looking for information about a type of object, such as a table or paragraph, use the table of contents to locate the MIF statements that describe the object.

MIF file layout

The following table lists the main statements in a MIF document file in the order that Adobe® FrameMaker® writes them. You must follow the same order that FrameMaker uses, with the exception of the macro statements and control statements, which can appear anywhere at the top level of a file. Each statement, except the `MIFFile` statement, is optional. Most main statements use substatements to describe objects and their properties.

Statement	Description
`MIFFile`	Labels the file as a MIF document file. The `MIFFile` statement is required and must be the first statement in the file.
Control statements	Establish the default units in a `Units` statement, the debugging setting in a `Verbose` statement, and comments in a `Comment` statement. These statements can appear anywhere at the top level as well as in some substatements.
Macro statements	Define macros with a `define` statement and read in files with an `include` statement. These statements can appear anywhere at the top level.
`ColorCatalog`	Describes document colors. The `ColorCatalog` statement contains `Color` statements that define each color and tag.
`ConditionCatalog`	Describes condition tags. The `ConditionCatalog` statement contains `Condition` statements that define each condition tag and its properties.
`BoolCondCatalog`	Describes Boolean Condition Expressions. The `BoolCondCatalog` statement contains `BoolCond` statements that define each Boolean condition expression with its show/hide properties.
`CombinedFontCatalog`	Describes combined fonts. The `CombinedFontCatalog` statement contains `CombinedFontDefn` statements that define each combined font and its component fonts.
`PgfCatalog`	Describes paragraph formats. The `PgfCatalog` statement contains `Pgf` statements that define the properties and tag for each paragraph format.
`ElementDefCatalog`	Defines the contents of the Element Catalog for a structured document. For more information, see , "MIF Statements for Structured Documents and Books."
`FmtChangeListCatalog`	Defines the contents of the Format Change List Catalog for a structured document. For more information, see , "MIF Statements for Structured Documents and Books."
`DefAttrValuesCatalog`	Defines the DefAttrValuesCatalog for a structured document. For more information, see , "MIF Statements for Structured Documents and Books."
`AttrCondExprCatalog`	Defines the AttrCondExprCatalog for a structured document. For more information, see , "MIF Statements for Structured Documents and Books."

Statement	Description
FontCatalog	Describes character formats. The FontCatalog statement contains Font statements that define the properties and tag for each character format.
RulingCatalog	Describes ruling styles for tables. The RulingCatalog statement contains Ruling statements that define the properties for each ruling style.
TblCatalog	Describes table formats. The TblCatalog statement contains TblFormat statements that define the properties and tag for each table format.
StyleCatalog	Describes object styles. The StyleCatalog statement contains Style statements that define the properties and tags for each object style.
KumihanCatalog	Contains the Kumihan tables that specify line composition rules for Japanese text.
Views	Describes color views for the document. The Views statement contains View statements that define which colors are visible in each color view.
VariableFormats	Defines variables.The VariableFormats statement contains VariableFormat statements that define each variable.
MarkerTypeCatalog	Defines a catalog of user-defined markers for the current document. The MarkerTypeCatalog statement contains MarkerTypeCatalog statements that specify each user-defined marker.
XRefFormats	Defines cross-reference formats. The XRefFormats statement contains XRefFormat statements that define each cross-reference format.
Document	Controls document features such as page size, margins, and column layout. Because the MIF interpreter assumes the same page defaults as the New command, this section is necessary only if you want to override those default settings.
BookComponent	Provides the setup information for files generated from the document. BookComponent statements describe the filename, filename suffix, file type, and paragraph tags or marker types to include.
InitialAutoNums	Provides a starting value for the autonumber series in a document.
Dictionary	Lists allowed words in the document.
AFrames	Describes all anchored frames in the document. The AFrames statement contains Frame statements that define the contents ID number of each anchored frame. Later in the MIF file, where the document contents are described, the MIF file must include an AFrame statement that corresponds to each Frame statement. The AFrame statement identifies where a specific anchored frame appears in a text flow; it need only supply the frame's ID number.
Tbls	Describes all tables in the document. The Tbls statement contains Tbl statements that define the contents of each table and its ID number. Later in the MIF file, where the document contents are described, the MIF file must include a short ATbl statement that corresponds to each Tbl statement. The ATbl statement identifies where a specific table appears in a text flow; it need only supply the table's ID number.
Page	Describes the layout of each page in the document. The description includes the layout of each page, the dimensions of the text frames, and the objects and other graphic frames on that page. A MIF file created by FrameMaker includes a Page statement for each page in the document, including the master pages. When you write an import filter, you can omit Page statements; the MIF interpreter repaginates the document as needed.
InlineComponentsInfo	Describes the mini table of contents (mini TOC) in the document. The InlineComponentsInfo statement contains InlineComponentInfo statement that define the properties of the mini TOC.

Statement	Description
TextFlow	Represents the actual text in the document. Within TextFlow statements, the text is expressed in paragraphs which in turn contain paragraph lines. Line endings of ParaLine statements are not significant because the MIF interpreter wraps the contents of ParaLine statements into paragraphs.

MIFFile statement

The MIFFile statement identifies the file as a MIF file. The MIFFile statement is required and must be the first line of the file with no leading white space.

Syntax

<MIFFile *version*> #*comment*	(Required) Identifies a MIF file

The *version* argument indicates the version number of the MIF language used in the file, and *comment* shows the name and version number of the program that generated the file. For example, a MIF file saved in FrameMaker (2015 release) begins with the following line:

```
<MIFFile 2015> # Generated by FrameMaker 12.0.2.366
```

MIF is compatible across versions, so a MIF interpreter can parse any MIF file. The results may sometimes differ from your intentions if a MIF file describes features that are not included in FrameMaker that reads the MIF file. For more information, see , "MIF Compatibility."

Comment statement

The Comment statement identifies an optional comment.

Syntax

<Comment *comment-text*>	Identifies a comment

Usage

Comments can appear within Comment statements, or they can follow a number sign (#). When it encounters a number sign, the MIF interpreter ignores all text until the end of the line, including angle brackets.

Because Comment statements can be nested within one another, the MIF interpreter examines all characters following an angle bracket until it finds the corresponding angle bracket that ends the comment.

```
<Comment - The following statements define the paragraph formats>
<Comment <These statements have been removed: <Font <FBold> <FItalic>>>>
```

The MIF interpreter processes number signs within Comment statements as normal comments, ignoring the remainder of the line.

```
<Comment - When a number sign appears within a <Comment> statement,
# the MIF interpreter ignores the rest of the characters in that
# line--including angle brackets < >.>
# End of <Comment> Statement.
```

Macro statements

MIF has two statements that allow you to define macros and include information from other files. Although these statements usually appear near the beginning of a MIF file, you need not put them in that position. However, the MIF interpreter does not interpret a macro that occurs before its definition.

define statement

The `define` statement creates a macro. When the MIF interpreter reads a MIF file, it replaces the macro name with its replacement text. A `define` statement can appear anywhere in a MIF file; however, the macro definition must appear before any occurrences of the macro name.

Syntax

`define (`*`name`*`, `*`replacement`*`)`	Creates a macro

Usage

Once a macro has been defined, you can use the macro name anywhere that the replacement text is valid. For example, suppose you define the following macro:

```
define (Bold, <Font <FWeight `Bold'>>)
```

When you use the macro in MIF statements, write `<Bold>`. The interpreter replaces `<Bold>` with `<Font <FWeight `Bold'>>`. Note that it retains the outer angle brackets in the replacement text.

Note that when you use a macro in a MIF file, you must enclose macro names in brackets to comply with the MIF syntax (for example, write `<Bold>` instead of `Bold`). The MIF parser requires these brackets to interpret the macro correctly.

include statement

The `include` statement reads information from other files. It is similar to an `#include` statement in a C program. When the MIF interpreter reads a MIF file, it replaces the `include` statement with the contents of the included file. An `include` statement can appear anywhere in a MIF file. However, make sure that the contents of the included file appear in a valid location when they are read into the MIF file.

Syntax

`include (`*`pathname`*`)`	Reads in a file

Usage

The *`pathname`* argument specifies a UNIX-style pathname, which uses a slash (`/`) to separate directory names (for example, `/usr/doc/template.mif`). For the Windows version of FrameMaker, use the following guideline for specifying absolute pathnames:

- For Windows versions, start an absolute pathname with the drive name. For example, to include the file `myfile.doc` from the directory `mydir` on the `c:` drive, specify the pathname `c:/mydir/myfile.doc`. Don't start an absolute path with a slash (/).

If you specify a relative pathname, the MIF interpreter searches for the file to include in the directory or folder that contains the file being interpreted. In UNIX versions of FrameMaker, the MIF interpreter also searches the `$FMHOME/fminit` and the `$FMHOME/fminit/filters` directories for a file with a relative pathname.

In general, you would use an `include` statement to read a header file containing `define` statements that a filter needs to translate a file. Isolate the data in a header file to simplify the process of changing important mappings. You can also use an `include` statement to read in a template file containing formatting information. Your application can then simply generate a document's text. For more information, see "Including template files" on page 45.

Track edited text

Reviewers can edit FrameMaker documents sent for review with the Track Text Edit feature enabled. In a MIF file, you can enable the Track Text Edit feature using the `DTrackChangesOn` statement. FrameMaker retains the Windows/Unix login name of the reviewer and a timestamp indicating the time of the edit in each of the edits. Before you accept all text edits, you can preview the final document with all the text edits or the text edits by a specific reviewer incorporated in the document. Alternatively, you can preview the original document without the text edits incorporated in the document. To preview how a document will appear if you accept all text edits or reject all text edits, use the `DTrackChangesPreviewState` statement.

Syntax

`<DTrackChangesOn` *`boolean`*`>`	Preserves the On/Off state of the Track Text Edit feature
`<DTrackChangesPreviewState` *`integer`*`>`	Preserves the preview state of the Track Text Edit feature The preview state can have one of the following values: **Preview Off:** `DTrackChangesPreviewState` set with the value `No` **Preview On Final:** `DTrackChangesPreviewState` set with the value `All` **Preview On Original:** `DTrackChangesPreviewState` set with the value `Yes`
`<DTrackChangesReviewerName` *`string`*`>`	The windows/unix login name of the reviewer whose edits are visible in the document The Show Reviewer Name popup menu lets you select the name of the reviewer whose changes you want to display in the document. The reviewer's name selected in the Show Reviewer Name popup menu appears in this tag. When you select All Users, this tag is empty.
`<ReviewerName` *`string`*`>`	The windows/unix login name of the reviewer who made a particular change
`<ReviewTimeInfo` *`string`*`>`	The time when an edit was made The number of seconds past after 00:00 hours, Jan 1, 1970 UTC

Conditional text

FrameMaker documents can contain conditional text. In a MIF file, the condition tags are defined by a `Condition` statement, which specifies whether the condition tag is hidden or shown. The condition tags for a document are stored in a `ConditionCatalog` statement.

Within the text flow, `Conditional` and `Unconditional` statements show where conditional text begins and ends.

ConditionCatalog statement

The `ConditionCatalog` statement defines the contents of the Condition Catalog. A MIF file can have only one `ConditionCatalog` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<ConditionCatalog`	
`<Condition…>`	Defines a condition tag (see "Condition statement," next)
`<Condition…>`	Additional statements as needed
…	
`>`	End of `ConditionCatalog` statement

Condition statement

The `Condition` statement defines the state of a condition tag and its condition indicators, which control how conditional text is displayed in the document window. The statement must appear in a `ConditionCatalog` statement. The property statements can appear in any order.

Syntax

`<Condition`	
`<CTag` *`string`*`>`	Condition tag string

`<CState` *`keyword`*`>`	Whether text with this tag is shown or hidden
	`keyword` can be one of:
	`CHidden`
	`CShown`

`<CStyle` *`keyword`*`>`	Format of text with this condition
	`keyword` can be one of:
	`CAsIs`
	`CUnderline`
	`CDoubleUnderline`
	`CStrike`
	`COverline`
	`CChangeBar`

`<CColor` *`tagstring`*`>`	Color for condition tag (see "ColorCatalog statement" on page 84)
`<CSeparation` *`integer`*`>`	Color for condition tag; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
`<CBackgroundColor` *`tagstring`*`>`	Background color of the conditional tag's text
`>`	End of `Condition` statement

Conditional and Unconditional statements

The `Conditional` statement marks the beginning of conditional text and the `Unconditional` statement marks the end. These statements must appear in a `Row` or `ParaLine` statement.

Syntax

`<Conditional`	Begin conditional text
`<InCondition` *tagstring*`>`	Specifies condition tag from Condition Catalog
`<InCondition` *tagstring*`>`	Additional statements as needed
...	
`>`	End of `Conditional` statement
`<Unconditional>`	Returns to unconditional state

System generated colors

FrameMaker will automatically generate new colors when multiple tags are applied on text. The `ColorTag` tag that is generated is named with the "fm_gen_" prefix and appended with a system-generated integer.

Boolean expressions

A Boolean expression is defined in a `BoolCond` statement.

BoolCondCatalog statement

You can create Boolean expressions by linking different conditional tags using Boolean operators. In a MIF file, Boolean condition expressions are defined using a `BoolCond` statement. The Boolean expressions for a document are stored in a `BoolCondCatalog` statement.

The `BoolCondCatalog` statement defines the contents of Boolean Expression Catalog for conditional text. A MIF file can have only one `BoolCondCatalog` statement, after Condition Catalog.

Syntax

`<BoolCondCatalog`	
`<BoolCond.........>`	Defines a Boolean expression
`<BoolCond.........>`	
`> #`	End of `BoolCondCatalog`

BoolCond statement

The `BoolCond` statement defines a new boolean expression, which is used to evaluate the show/hide state of conditional text. Statement must appear in `BoolCondCatalog` statement. The property statement can appear in any order.

Syntax

`<BoolCond`	
`<BoolCondTag string>`	Tag name used for Boolean expressions.

<BoolCondExpr string>	Boolean expression used for show/hide evaluation of conditional text. (OR, NOT, and AND are the operators and condition tags are operands within a quoted string) For example, "Comment" OR "Tag1".
<BoolCondState string>	Indicates whether the evaluation of showing or hiding conditional text is based on this expression. The string must contain one of the following values: • 'Active' • 'Inactive'
> #	End of BoolCond

Filter By Attribute

Elements in a structured document can have one or more attributes associated with them. Using FrameMaker, you can filter a structured document based on the value of these attributes.

All MIF 8 documents contain a catalog of predefined attribute values. If no values are defined, the catalog remains empty. Each definition in a catalog includes an attribute tag (AttributeTag) and the corresponding list of values (AttributeValue).

DefAttrValuesCatalog statement

The DefAttrValuesCatalog statement is used to define the contents of the Defined Attribute Values catalog. A MIF file can contain one DefAttrValuesCatalog statement only.

Syntax

<DefAttrValuesCatalog	
<DefAttrValues.....>	Defines an attribute and its corresponding values
<DefAttrValues.....>	Additional statements, as required.
> #	End of DefAttrValuesCatalog

All MIF 8 documents contain a catalog of predefined filters.

DefAttrValues statement

The DefAttrValues statement is used to define a set of attributes with relevant values.

Syntax

<DefAttrValues	
<AttributeTag *string*>	Attribute Name
<AttributeValue *string*>	Attribute Value
<AttributeValue *string*>	Additional attribute values, as required.
> #	End of DefAttrValues

AttrCondExprCatalog statement

The `AttrCondExprCatalog` statement is used to define the contents of the Attribute Expression catalog. A MIF file can contain one `AttrCondExprCatalog` statement only.

Syntax

`<AttrCondExprCatalog`	
`  <AttrCondExpr.....>`	Defines a filter
`  <AttrCondExpr.....>`	Additional filters, as required.
`> #`	End of AttrCondExprCatalog

AttrCondExpr statement

The `AttrCondExpr` statement is used to define a set of attributes with values.

Syntax

`<AttrCondExpr`	
`  <AttrCondExprTag string>`	Expression Tag string
`  <AttrCondExprStr string>`	Expression string

`  <AttrCondState  string>`	Indicates whether the `AttrCondExpr` is applied to the document.
	The string must have one of the following values:
	`'Active'`
	`'Inactive'`

`> #`	End of AttrCondExpr

Paragraph formats

A paragraph format is defined in a `Pgf` statement. Paragraph formats can be defined locally or stored in the Paragraph Catalog, which is defined by a `PgfCatalog` statement.

PgfCatalog statement

The `PgfCatalog` statement defines the contents of the Paragraph Catalog. A MIF file can have only one `PgfCatalog` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<PgfCatalog`	
`  <Pgf...>`	Defines a paragraph format (see "Pgf statement" on page 62)
`  <Pgf...>`	Additional statements as needed
`  ...`	
`>`	End of `PgfCatalog` statement

Usage

If you don't include a PgfCatalog statement, the MIF interpreter uses the paragraph formats defined in NewTemplate. (For information on defaults specified in templates, see page 3.) If you include PgfCatalog, paragraph formats in the MIF file replace default formats. The MIF interpreter does not add your paragraph format to the default Paragraph Catalog, although it provides default values for unspecified properties in a paragraph format (see "Creating and applying paragraph formats" on page 12).

Pgf statement

The Pgf statement defines a paragraph format. Pgf statements can appear in many statements; the statement descriptions show where Pgf can be used.

The Pgf statement contains substatements that set the properties of a paragraph format. Most of these properties correspond to those in the Paragraph Designer. Properties can appear in any order within a Pgf statement, with the following exception: the PgfNumTabs statement must appear before any TabStop statements.

Syntax

Basic properties	
<Pgf	Begin paragraph format
<PgfTag *tagstring*>	Paragraph tag name
<PgfUseNextTag *boolean*>	Turns on following paragraph tag feature
<PgfNextTag *tagstring*>	Tag name of following paragraph
<PgfFIndent *dimension*>	First line left margin, measured from left side of current text column
<PgfFIndentRelative *boolean*>	Used for structured documents only
<PgfFIndentOffset *dimension*>	Used for structured documents only
<PgfLIndent *dimension*>	Left margin, measured from left side of current text column
<PgfRIndent *dimension*>	Right margin, measured from right side of current text column
<PgfAlignment *keyword*>	Alignment within the text column *keyword* can be one of: LeftRight Left Center Right
<PgfDir *keyword*>	Direction of the paragraph. *keyword* can be one of: LTR – The direction of the paragraph is set to left to right RTL – The direction of the paragraph is set to right to left. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right, then INHERITLTR is assigned to PgfDir. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left, then INHERITRTL is assigned to PgfDir.
<PgfSpBefore *dimension*>	Space above paragraph
<PgfSpAfter *dimension*>	Space below paragraph

<PgfLineSpacing *keyword*>	Amount of space between lines in paragraph measured from baseline to baseline *keyword* can be one of: Fixed (default font size) Proportional (largest font in line)
<PgfLeading *dimension*>	Space below each line in a paragraph
<PgfNumTabs *integer*>	Number of tabs in a paragraph The statement is not required for input files; the MIF interpreter calculates the number of tabs. If it does appear, it must appear before any TabStop statements; otherwise, the MIF interpreter ignores the tab settings.
<TabStop	Begin definition of tab stop; the following property statements can appear in any order, but must appear within a TabStop statement
<TSX *dimension*>	Horizontal position of tab stop
<TSType *keyword*>	Tab stop alignment *keyword* can be one of: Left Center Right Decimal
<TSLeaderStr *string*>	Tab stop leader string (for example, `.')
<TSDecimalChar *integer*>	Align decimal tab around a character by ASCII value; in UNIX versions, type man ascii in a UNIX window for a list of characters and their corresponding ASCII values
>	End of TabStop statement
<TabStop...>	Additional statements as needed
Default font properties	
<PgfFont...>	Default font (see page 67)
Pagination properties	
<PgfPlacement *keyword*>	Vertical placement of paragraph in text column *keyword* can be one of: Anywhere ColumnTop PageTop LPageTop RPageTop
<PgfPlacementStyle *keyword*>	Placement of side heads, run-in heads, and paragraphs that straddle text columns *keyword* can be one of: Normal RunIn SideheadTop SideheadFirstBaseline SideheadLastBaseline Straddle StraddleNormalOnly See page 66

<PgfRunInDefaultPunct *string*>	Default punctuation for run-in heads
<PgfWithPrev *boolean*>	Yes keeps paragraph with previous paragraph
<PgfWithNext *boolean*>	Yes keeps paragraph with next paragraph
<PgfBlockSize *integer*>	Widow/orphan lines
Numbering properties	
<PgfAutoNum *boolean*>	Yes turns on autonumbering
<PgfNumFormat *string*>	Autonumber formatting string
<PgfNumberFont *tagstring*>	Tag from Character Catalog
<PgfNumAtEnd *boolean*>	Yes places number at end of line, instead of beginning
Advanced properties	
<PgfHyphenate *boolean*>	Yes turns on automatic hyphenation
<HyphenMaxLines *integer*>	Maximum number of consecutive lines that can end in a hyphen
<HyphenMinPrefix *integer*>	Minimum number of letters that must precede hyphen
<HyphenMinSuffix *integer*>	Minimum number of letters that must follow a hyphen
<HyphenMinWord *integer*>	Minimum length of a hyphenated word
<PgfLetterSpace *boolean*>	Spread characters to fill line
<PgfMinWordSpace *integer*>	Minimum word spacing (as a percentage of a standard space in the paragraph's default font)
<PgfOptWordSpace *integer*>	Optimum word spacing (as a percentage of a standard space in the paragraph's default font)
<PgfMaxWordSpace *integer*>	Maximum word spacing (as a percentage of a standard space in the paragraph's default font)

<PgfLanguage *keyword*>	Language to use for spelling and hyphenation. Note that FrameMaker writes this statement so MIF files can be opened in older versions of FrameMaker. However, the language for a paragraph format or character format is now properly specified in the PgfFont and Font statements (see page 67) *keyword* can be one of: NoLanguage USEnglish UKEnglish German SwissGerman AustriaGerman German1996 SwissGerman1996 French CanadianFrench Spanish Catalan Italian Portuguese Brazilian Danish Dutch Norwegian Nynorsk Finnish Swedish Japanese TraditionalChinese SimplifiedChinese Korean Arabic Hebrew
<PgfTopSeparator *string*>	Name of reference frame (from reference page) to put above paragraph
<PgfTopSepAtIndent *boolean*>	Used for structured documents only
<PgfTopSepOffset *dimension*>	Used for structured documents only
<PgfBoxColor *string*>	The background color for the entire box that surrounds a paragraph.
<PgfBotSeparator *string*>	Name of reference frame (from reference page) to put below paragraph
<PgfBotSepAtIndent *boolean*>	Used for structured documents only
<PgfBotSepOffset *dimension*>	Used for structured documents only
Table cell properties	
<PgfCellAlignment *keyword*>	Vertical alignment for first paragraph in a cell *keyword* can be one of: Top Middle Bottom
<PgfCellMargins *L T R B*>	Cell margins for first paragraph in a cell
<PgfCellLMarginFixed *boolean*>	Yes means left cell margin is added to TblCellMargins; No means left cell margin overrides TblCellMargins
<PgfCellTMarginFixed *boolean*>	Yes means top cell margin is added to TblCellMargins; No means top cell margin overrides TblCellMargins

<PgfCellRMarginFixed *boolean*>	Yes means right cell margin is added to TblCellMargins; No means right cell margin overrides TblCellMargins
<PgfCellBMarginFixed *boolean*>	Yes means bottom cell margin is added to TblCellMargins; No means width of bottom cell margin overrides TblCellMargins
Miscellaneous properties	
<PgfLocked *boolean*>	Yes means the paragraph is part of a text inset that obtains its formatting properties from the source document. See page 66
<PgfAcrobatLevel *integer*>	Level at which the paragraph is shown in an outline of Acrobat Bookmarks; 0 indicates that the paragraph does not appear as a bookmark

Usage

Within a PgfCatalog statement, the PgfTag statement assigns a tag to a paragraph format. To apply a paragraph format from the Paragraph Catalog to the current paragraph, use the PgfTag statement in a ParaLine statement.

If the PgfTag statement within a text flow does not match a format in the Paragraph Catalog, then the Pgf statement makes changes to the current paragraph format. That is, a Pgf statement after PgfTag specifies how the paragraph differs from the format in the catalog.

If a document has side heads, indents and tabs are measured from the text column, not the side head. In a table cell, tab and indent settings are measured from the cell margins, not the cell edges.

Usage of some aspects of the Pgf statement is described in the following sections.

Paragraph placement across text columns and side heads

The PgfPlacementStyle statement specifies the placement of a paragraph across text columns and side heads in a text frame:

- If a paragraph spans across all columns and side heads, the PgfPlacementStyle statement is set to Straddle.
- If a paragraph spans across all columns, but not across the side heads in a text frame, the PgfPlacementStyle statement is set to StraddleNormal.

Locked paragraphs and text insets

The PgfLocked statement does not correspond to any setting in the Paragraph Designer. The statement is used for text insets that retain formatting information from the source document.

If the <PgfLocked Yes> statement appears in a specific paragraph, that paragraph is part of a text inset that retains formatting information from the source document. The paragraph is not affected by global formatting performed on the document.

If the <PgfLocked No> statement appears in a specific paragraph, that paragraph is not part of a text inset, or is part of a text inset that reads formatting information from the current document. The paragraph is affected by global formatting performed on the document.

For more information about text insets, see "Text insets (text imported by reference)" on page 138.

Character formats

A character format is defined by a PgfFont or a Font statement. Character formats can be defined locally or they can be stored in the Character Catalog, which is defined by a FontCatalog statement.

FontCatalog statement

The `FontCatalog` statement defines the contents of the Character Catalog. A document can have only one `FontCatalog` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<FontCatalog`	
`<Font…>`	Defines a character format (see "PgfFont and Font statements," next)
`<Font…>`	Additional statements as needed
…	
`>`	End of `FontCatalog` statement

PgfFont and Font statements

The `PgfFont` and `Font` statements both define character formats. The `PgfFont` statement must appear in a `Pgf` statement. The `Font` statement must appear in a `FontCatalog`, `Para`, or `TextLine` statement.

New statements have been added to the PgfFont and Font statements to express combined fonts in FrameMaker documents. For more information, see "Combined Fonts" on page 214.

Syntax

`<PgfFont` | `Font`	
`<FTag` *`tagstring`*`>`	Character format tag name
Font name	
`<FFamily` *`string`*`>`	Name of font family
`<FAngle` *`string`*`>`	Name of angle, such as `Oblique`
`<FWeight` *`string`*`>`	Name of weight, such as `Bold`
`<FVar` *`string`*`>`	Name of variation, such as `Narrow`
`<FPostScriptName` *`string`*`>`	Name of font when sent to PostScript printer (see "Font name" on page 70)
`<FPlatformName` *`string`*`>`	Platform-specific font name, only read by the Windows version (see page 71)

Font language	
<FLanguage *keyword*>	Language to use for spelling and hyphenation *keyword* can be one of: NoLanguage USEnglish UKEnglish German SwissGerman French CanadianFrench Spanish Catalan Italian Portuguese Brazilian Danish Dutch Norwegian Nynorsk Finnish Swedish Japanese TraditionalChinese SimplifiedChinese Korean Arabic Hebrew
Font encoding	
<FEncoding *keyword*>	Specifies the encoding for this font. This is to specify the encoding for a double-byte font. If not present, the default is Roman. *keyword* can be one of these: FrameRoman JISX0208.ShiftJIS BIG5 GB2312-80.EUC KSC5601-1992
Font size, color, and width	
<FSize *dimension*>	Size, in points only (or in Q on a Japanese system)
<FColor *tagstring*>	Font color (see "ColorCatalog statement" on page 84)
<FSeparation *integer*>	Font color; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
<FStretch *percent*>	The amount to stretch or compress the font, where 100% means no change
<FBackgroundColor *tagstring*>	Background color of the paragraph text.
Font style	
<FUnderlining *keyword*>	Turns on underlining and specifies underlining style *keyword* can be one of: FNoUnderlining FSingle FDouble FNumeric

<FOverline *boolean*>	Turns on overline style
<FStrike *boolean*>	Turns on strikethrough style
<FChangeBar *boolean*>	Turns on the change bar
<FPosition *keyword*>	Specifies subscript and superscript characters; font size and position relative to baseline determined by Document substatements (see page 94) *keyword* can be one of: FNormal FSuperscript FSubscript
<FOutline *boolean*>	Turns on outline style
<FShadow *boolean*>	Turns on shadow style
<FPairKern *boolean*>	Turns on pair kerning
<FCase *keyword*>	Applies capitalization style to string *keyword* can be one of: FAsTyped FSmallCaps FLowercase FUppercase
Kerning information	
<FDX *percent*>	Horizontal kern value for manual kerning expressed as percentage of an em; positive value moves characters right and negative value moves characters left
<FDY *percent*>	Vertical kern value for manual kerning expressed as percentage of an em; positive value moves characters down and negative value moves characters up
<FDW *percent*>	Spread value for space between characters expressed as percentage of an em; positive value increases the space and negative value decreases the space
<FTsume *boolean*>	Yes turns on Tsume (variable width rendering) for Asian characters
Filter statements	Valid when text properties are applied to a file imported into FrameMaker
<FPlain *boolean*>	Used only by filters
<FBold *boolean*>	Used only by filters
<FItalic *boolean*>	Used only by filters
Miscellaneous information	
<FLocked *boolean*>	Yes means the font is part of a text inset that obtains its formatting properties from the source document
>	End of PgfFont or Font statement

Usage

Use PgfFont within a Pgf statement to override the default font for the paragraph. Use Font within a FontCatalog statement to define a font or in a Para statement to override the default character format. Substatements in the Font and PgfFont statements are optional. Like the Pgf substatements, Font substatements reset the current font.

When the MIF interpreter reads a `Font` statement, it continues using the character format properties until it either reads another `Font` statement or reads the end of the `Para` statement. You can set the character format back to its previous state by providing an empty `FTag` statement. A `Font` statement that does not supply all property substatements inherits the current font state for those properties not supplied.

For more information about creating and applying character formats in a MIF file, see "Creating and applying character formats" on page 24. For more information about character formats in general, see your user's manual.

Usage of some aspects of the `PgfFont` and `Font` statements is described in the following sections.

Locked fonts and text insets

The `FLocked` statement does not correspond to any setting in the Character Designer. The statement is used for text insets that retain formatting information from the source document.

If the `<FLocked Yes>` statement appears in a specific character format, that character format is part of a text inset that retains formatting information from the source document. The character format is not affected by global formatting performed on the document.

If the `<FLocked No>` statement appears in a specific character format, either that character format is not part of a text inset, or that character format is part of a text inset that reads formatting information from the current document. The character format is affected by global formatting performed on the document.

For more information about text insets, see "Text insets (text imported by reference)" on page 138.

Font name

When a `PgfFont` or `Font` statement includes all of the family, angle, weight, and variation properties, FrameMaker identifies the font in one or more of the following ways:

- The statement `FPlatformName` specifies a font name that uniquely identifies the font on a specific platform.
- The statements `FFamily`, `FAngle`, `FWeight`, and `FVar` specify how FrameMaker stores font information internally.
- The statement `FPostScriptName` specifies the name given to a font when it is sent to a PostScript printer (specifically, the name that would be passed to the PostScript `FindFont` operator before any font coordination operations). The PostScript name is unique for all PostScript fonts, but may not be available for fonts that have no PostScript version.

For complete font specifications, FrameMaker always writes the `FFamily`, `FAngle`, `FWeight`, `FVar`, and `FPostScriptName` statements. In addition, the Windows version of FrameMaker also writes the `FPlatformName` statement. A UNIX version of FrameMaker ignores `FPlatformName`.

When FrameMaker reads a MIF file that includes more than one way of identifying a font, it checks the font name in the following order:

1. Platform name
2. Combination of family, angle, weight, and variation properties
3. PostScript name

If you are writing filters to generate MIF, you do not need to use all three methods. You should always specify the PostScript name, if it is available. You should use the platform name only if your filter will be run on a specific platform. A filter running on a specific platform can easily find and write out the platform name, but the name cannot be used on other platforms.

Font encoding

The `<FEncoding>` statement specifies which encoding to use for a font. The default is Roman, or standard 7-bit encoding. If this statement is not included for a font, 7-bit encoding is assumed.

This statement takes precedence over all other font attributes. For example, if the document includes a font with <FEncoding `JISX0208.ShiftJIS'>, but that font family is not available on the user's system, then the text will appear in some other font on the system that uses Japanese encoding. If there is no Japanese encoded font on the system, the text appears in Roman encoding and the user will see garbled characters.

FPlatformName statement

The <FPlatformName *string*> statement provides a platform-specific ASCII string name that uniquely identifies a font for a particular platform. The *string* value consists of several fields separated by a period.

Windows: The Windows platform name has the following syntax:

<FPlatformName W.*FaceName.ItalicFlag.Weight.Variation*>

W	Platform designator
FaceName	Windows face name (for more information, see your Windows documentation)
ItalicFlag	Whether font is italic; use one of the following flags: I (Italic) R (Regular)
Weight	Weight classification, for example 400 (regular) or 700 (bold)
Variation	Optional variation, for example Narrow

The following statements are valid representations of the Windows font Helvetica Narrow Bold Oblique:

```
<FPlatformName W.Helvetica-Narrow.I.700>
<FPlatformName W.Helvetica.I.700.Narrow>
```

Object styles

An object style is defined by a Style statement. Object styles can be defined locally or they can be stored in the Object Style catalog, which is defined by a StyleCatalog statement.

StyleCatalog statement

The StyleCatalog statement defines the object styles. A document can have only one StyleCatalog statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

<StyleCatalog	
<Style	Defines a character format (see "PgfFont and Font statements," next)
>	End of StyleCatalog statement

Style statement

The Style statement defines the object style properties. A document can have only one StyleCatalog statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

<Style	
<StyleTag *string*>	The name of the object style.
<Pen *integer*>	Pen pattern for lines and edges (see "Values for Pen and Fill statements" on page 107)
<PenWidth *dimension*>	Line and edge thickness
<ObTint *percentage*>	Applies a tint to the object color; 100% is equivalent to the pure object color and 0% is equivalent to no color at all
<DashedPattern	
<DashedStyle *keyword*>	Specifies whether object is drawn with a dashed or a solid line *keyword* can be one of: Solid Dashed
<HeadCap *keyword*>	Type of head cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<TailCap *keyword*>	Type of tail cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<ArrowStyle ...>	See "ArrowStyle statement" on page 116.
<RunaroundGap *dimension*>	Space between the object and the text flowing around the object; must be a value between 0.0 and 432.0 points.
<Angle *integer*>	Angle of rotation in degrees: 0, 90, 180, 270
<OffsetTop *dimension*>	Offset from top
<OffsetLeft *dimension*>	Offset from left
<SizeWidth *dimension*>	Width of text
<SizeHeight *dimension*>	Height of text
<AFrameDir *keyword*>	Controls the direction of the anchored frame. *keyword* can be one of: LTR – Set the direction for the anchored frame to left to right. RTL – Set the direction for the anchored frame to right to left. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right then INHERITLTR is assigned to AFrameDir. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left then INHERITRTL is assigned to AFrameDir.
<TFrameNumColumns *integer*>	Number of columns in the text frame (1-10)

<TFrameColumnGap *integer*>	Space between columns in the text frame (0"-50")
<TFrameShRoom *boolean*>	`Yes` gives room for side heads
<TFrameShWidth dimension>	Side head width
<TFrameShGap dimension>	Gap between side head and body text areas
<TFrameAutoconnect boolean>	`Yes` adds text frames as needed to extend flows
<TFramePostscript boolean>	`Yes` identifies text in the flow as printer code
<TFrameColumnBalance boolean>	`Yes` means columns in the text frame are automatically adjusted to the same height
<TFrameDir *keyword*>	Controls the direction of the text frame and its child objects. *keyword* can be one of: `LTR` – Set the direction of the text flow object to left to right. The text flow propagates its direction to all child objects that derive their direction from the text flow object. `RTL` – Set the direction of the text flow object to right to left. The text flow propagates its direction to all child objects that derive their direction from the text flow object. `INHERITLTR` – Derive the direction from the parent object. If it resolves to left to right, then INHERITLTR is assigned to `TFrameDir`. `INHERITRTL` – Derive the direction from the parent object. If it resolves to right to left, then INHERITRTL is assigned to `TFrameDir`.
<TLineDir *keyword*>	Controls the direction in which the text line is drawn. *keyword* can be one of: `LTR` – Set the direction for the text line object to left to right. `RTL` – Set the direction for the text line object to right to left. `INHERITLTR` – Derive the direction from the parent object. If it resolves to left to right then `INHERITLTR` is assigned to `TLineDir`. `INHERITRTL` – Derive the direction from the parent object. If it resolves to right to left then INHERITRTL is assigned to `TLineDir`.
<ATheta dimension>	Start angle
<ADTheta dimension>	Arc angle length
<InsetScaling dimension>	Scaling of the inset
<InsetOpacity integer>	Opacity value defined for an object.
<EquationBreak dimension>	Set line-width after which the equation breaks to the next line
<MathMLStyleDpi integer>	Scaling value using which bitmap file is imported
<MathMLStyleComposeDpi integer>	Scaling value using which bitmap file is created

<MathMLStyleFontSize *dimension*>	Font size of the MathML to which the style is applied
<MathMLStyleInline *boolean*>	Yes places the equation inline with the paragraph text

Line numbers

FrameMaker documents can have the line numbers displayed for assisting in the reviewing process. Multiple contributors to the document can refer to the content using the Page number and then line number. The following are the statements relevant to line numbers:

Syntax

<DLineNumGap *dimension*>	The width of the line number field.
<DLineNumRestart *boolean*>	Setting this property to Yes restarts the line numbering to 1 for each page of a document.
<DLineNumShow *boolean*>	Setting this property to Yes displays the line numbers.
<DLineNumFontFam *string*>	Name of the font family for the line numbers.
<DLineNumSize *dimension*>	Size of the line number text, in points.
<DLineNumColor *tagstring*>	Color of the line number text.

Tables

Table formats are defined by a TblFormat statement. Table formats can be locally defined or they can be stored in a Table Catalog, which is defined by a TblCatalog statement. The ruling styles used in a table are defined in a RulingCatalog statement.

In a MIF file, all document tables are contained in one Tbls statement. Each table instance is contained in a Tbl statement. The ATbl statement specifies where each table instance appears in the text flow.

TblCatalog statement

The TblCatalog statement defines the Table Catalog. A document can have only one TblCatalog statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

<TblCatalog	
<TblFormat...>	Defines a table format (see "TblFormat statement," next)
<TblFormat...>	Additional statements as needed
...	

>	End of `TblCatalog` statement

TblFormat statement

The `TblFormat` statement defines the format of a table. A `TblFormat` statement must appear in a `TblCatalog` or in a `Tbl` statement. A `TblFormat` statement contains property substatements that define a table's properties. Table property statements can appear in any order.

Syntax

Basic properties	
`<TblFormat`	
`<TblTag` *tagstring*`>`	Table format tag name
`<TblLIndent` *dimension*`>`	Left indent for the table relative to the table's containing text column; has no effect on right-aligned tables
`<TblRIndent` *dimension*`>`	Right indent for the table relative to the table's containing text column; has no effect on left-aligned tables
`<TblSpBefore` *dimension*`>`	Space above table
`<TblSpAfter` *dimension*`>`	Space below table

`<TblAlignment` *keyword*`>`	Horizontal alignment within text column or text frame
	keyword can be one of:
	`Left`
	`Center`
	`Right`
	`Inside`
	`Outside`
	See page 78

`<TblPlacement` *keyword*`>`	Vertical placement of table within text column
	keyword can be one of:
	`Anywhere`
	`Float`
	`ColumnTop`
	`PageTop`
	`LPageTop`
	`RPageTop`

`<TblBlockSize` *integer*`>`	Widow/orphan rows for body rows
`<TblCellMargins` *L T R B*`>`	Left, top, right, bottom default cell margins

`<TblTitlePlacement` *keyword*`>`	Table title placement
	keyword can be one of:
	`InHeader`
	`InFooter`
	`None`

`<TblTitlePgf1`	Paragraph format of title for a new table created with the table format
`<PgfTag` *tagstring*`>`	Applies format from Paragraph Catalog
`<Pgf…>`	Overrides Paragraph Catalog format as needed (see page 62)

>	End of TblTitlePgf1 statement
<TblTitleGap *dimension*>	Gap between title and top or bottom row
<TblNumByColumn *boolean*>	Autonumber paragraphs in cells; Yes numbers down each column and No numbers across each row
<TblDir *keyword*>	Direction of the table. *keyword* can be one of: LTR – The direction of the table is set to left to right. RTL – The direction of the table is set to right to left. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right, then INHERITLTR is assigned to TblDir. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left, then INHERITRTL is assigned to TblDir.
Ruling properties	
<TblColumnRuling *tagstring*>	Ruling style for most columns; value must match a ruling style name specified in the RulingCatalog statement
<TblXColumnNum *integer*>	Number of column with a right side that uses the TblXColumnRuling statement
<TblXColumnRuling *tagstring*>	Ruling style for the right side of column TblXColumnNum
<TblBodyRowRuling *tagstring*>	Default ruling style for most body rows
<TblXRowRuling *tagstring*>	Exception ruling style for every nth body row
<TblRulingPeriod *integer*>	Number of body rows after which TblXRowRuling should appear
<TblHFRowRuling *tagstring*>	Ruling style between rows in the heading and footing
<TblSeparatorRuling *tagstring*>	Ruling style for rule between the last heading row and first body row, and also between the last body row and the first footing row
<TblLRuling *tagstring*>	Left outside table ruling style
<TblBRuling *tagstring*>	Bottom outside table ruling style
<TblRRuling *tagstring*>	Right outside table ruling style
<TblTRuling *tagstring*>	Top outside table ruling style
<TblLastBRuling *boolean*>	Yes means draw bottom rule on the last sheet only; No means draw rule on the bottom of every sheet
Shading properties	
<TblHFFill *integer*>	Default fill pattern for table heading and footing (see page 113)
<TblHFColor *tagstring*>	Default color for table heading and footing (see page 85)
<TblHFSeparation *integer*>	Default color for table heading and footing; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
<TblBodyFill *integer*>	Default fill pattern for body cells (see page 113)
<TblBodyColor *tagstring*>	Default color for body cells (see page 85)
<TblBodySeparation *integer*>	Default color for body cells; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)

<TblShadeByColumn *boolean*>	Yes specifies column shading; No specifies body row shading
<TblShadePeriod *integer*>	Number of consecutive columns/rows that use TblBodyFill
<TblXFill *integer*>	Exception fill pattern for columns or body rows (see page 113)
<TblXColor *tagstring*>	Exception color for columns or body rows (see page 85)
<TblXSeparation *integer*>	Exception color for columns or body rows; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
<TblAltShadePeriod *integer*>	Number of consecutive columns/rows that use TblXFill; exception columns/rows alternate with default body columns/rows to form a repeating pattern
Column properties	
<TblWidth *dimension*>	Not generated by FrameMaker, but can be used by filters to determine table width
<TblColumn	Each table must have at least one TblColumn statement; a column without a statement uses the format of the rightmost column
<TblColumnNum *integer*>	Column number; columns are numbered from left to right starting at 0
<TblColumnWidth *dimension*>	Width of column. See page 82
<TblColumnWidthP *integer*>	Not generated by FrameMaker, but a temporary column width when filtering proportionally-spaced tables from another application; converted to a fixed width when read in (see page 82)
<TblColumnWidthA *W W*>	Not generated by FrameMaker, but a width based on a cell width, for filters only; converted into a fixed width when read in. First value is minimum width; second value is maximum width. Values limit the range of a computed column width, and are usually set to a wide range (see page 82).
<TblColumnH	Default paragraph format for the column's heading cells in new tables
<TableColumn	If the table column is conditionalized, the conditional properties are specified in the TableColumn property.
<Conditional	Specifies that the column is conditional.
<InCondition tagstring>	Applies the specified conditional tag to the column.
> '	End of Conditional statement.
>	End of end of TableColumn statement.
<PgfTag *tagstring*>	Applies format from Paragraph Catalog
<Pgf...>	Overrides Paragraph Catalog format as needed (see page 62)
>	End of TblColumnH statement
<TblColumnBody	Default paragraph format for the column's body cells in new tables
<PgfTag *tagstring*>	Applies format from Paragraph Catalog
<Pgf...>	Overrides Paragraph Catalog format as needed (see page 62)
>	End of TblColumnBody statement
<TblColumnF	Default paragraph format for the column's footing cells in new tables

<PgfTag *tagstring*>	Applies format from Paragraph Catalog
<Pgf...>	Overrides Paragraph Catalog format as needed (see page 62)
>	End of TblColumnF statement
>	End of TblColumn statement
<TblColumn...>	More TblColumn statements as needed, one per column
...	
New table properties	
<TblInitNumColumns *integer*>	Number of columns for new table
<TblInitNumHRows *integer*>	Number of heading rows for new table
<TblInitNumBodyRows *integer*>	Number of body rows for new tables
<TblInitNumFRows *integer*>	Number of footing rows for new tables
Miscellaneous properties	
<TblLocked *boolean*>	Yes means the table is part of a text inset that obtains its formatting properties from the source document
>	End of TblFormat statement

Usage

The basic properties, ruling properties, and shading properties correspond to settings in the Table Designer. The *tagstring* value specified in any ruling substatement (such as TblColumnRuling) must match a ruling tag defined in the RulingCatalog statement (see page 83). The *tagstring* value specified in any color substatement (such as TblBodyColor) must match a color tag defined in the ColorCatalog statement (see page 84).

Usage of some of the aspects of the TblFormat statement is described in the following sections.

Alignment of tables

The horizontal alignment of a table within a text column or text frame is specified by the TblAlignment statement:

- If the table is aligned with the left, center, or right side of a text column or text frame, the TblAlignment statement is set to Left, Center, or Right, respectively.
- If the table is aligned with the closer edge or farther edge of a text frame (closer or farther relative to the binding of the book), the TblAlignment statement is set to Inside or Outside, respectively.

Locked tables and text insets

The TblLocked statement does not correspond to any setting in the Table Designer. The statement is for text insets that retain formatting information from the source document.

If the <TblLocked Yes> statement appears in a specific table, that table is part of a text inset that retains formatting information from the source document. The table is not affected by global formatting performed on the document.

If the <TblLocked No> statement appears in a specific table, that table is not part of a text inset or is part of a text inset that reads formatting information from the current document. The table is affected by global formatting performed on the document.

For details about text insets, see "Text insets (text imported by reference)" on page 138.

Tbls statement

The `Tbls` statement lists the contents of each table in the document. A document can have only one `Tbls` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<Tbls`	Beginning of tables list
`<Tbl…>`	Defines a table instance (see "Tbl statement," next)
`<Tbl…>`	Additional statements as needed
…	
`>`	End of `Tbls` statement

Tbl statement

The `Tbl` statement contains the contents of a table instance. It must appear in a `Tbls` statement.

Each `Tbl` statement is tied to a location in a text flow by the ID number in a `TblID` statement. Each `Tbl` statement has an associated `ATbl` statement within a `ParaLine` statement that inserts the table in the flow. The `Tbl` statement must appear before the `ATbl` statement that refers to it. Each `Tbl` statement can have only one associated `ATbl` statement, and vice versa. For more information about the `ATbl` statement, see "ParaLine statement" on page 133.

Syntax

`<Tbl`	
`<TblID` *`ID`*`>`	Table ID number
`<TblTag` *`tagstring`*`>`	Applies format from Table Catalog
`<TblFormat…>`	Overrides Table Catalog format as needed (see page 75)
Table columns	
`<TblNumColumns` *`integer`*`>`	Number of columns in the table
`<TblColumnWidth` *`dimension`*`>`	Width of first column
`<TblColumnWidth` *`dimension`*`>`	Width of second column
…	Width of remaining columns as needed
`<EqualizeWidths`	Makes specified columns the same width as the widest column (for filters only, see page 82)
`<TblColumnNum` *`integer`*`>`	First column
`<TblColumnNum` *`integer`*`>`	More columns as needed
`>`	End of `EqualizeWidths` statement
Table title	
`<TblTitle`	Begin definition of table title
`<TblTitleContent`	Table title's content, represented in one or more `Para` statements
`<Notes…>`	Footnotes for table title (see page 131)

<Para…>	Title text (see page 132)
<Para…>	Additional statements as needed
…	
>	End of TblTitleContent statement
>	End of TblTitle statement
Table rows	
<TblH	Table heading rows; omit if no table headings
<Row…>	See “Row statement,” next
<Row…>	Additional statements as needed
…	
>	End of TblH statement
<TblBody	Table body rows
<Row…>	See “Row statement,” next
<Row…>	Additional statements as needed
…	
>	End of TblBody statement
<TblF	Table footing rows; omit if no table footing
<Row…>	See “Row statement,” next
<Row…>	Additional statements as needed
…	
>	End of TblF statement
>	End of Tbl statement

Usage

The table column statements specify the actual width of the table instance columns. They override the column widths specified in the TblFormat statement.

Row statement

A Row statement contains a list of cells. It also includes row properties as needed. The statement must appear in a Tbl statement.

Syntax

<Row	
<Conditional…>	Specifies conditional row (row is unconditional if the statement is omitted)
<RowWithNext *boolean*>	Keep with next body row

<RowWithPrev *boolean*>	Keep with previous body row
<RowMinHeight *dimension*>	Minimum row height
<RowMaxHeight *dimension*>	Maximum row height
<RowHeight *dimension*>	Row height
<RowPlacement *keyword*>	Row placement *keyword* can be one of: Anywhere ColumnTop LPageTop RPageTop PageTop
<Cell…>	Each Row statement contains one Cell statement for each column (see "Cell statement," next)
<Cell…>	Additional statements as needed
…	
>	End of Row statement

Usage

Each Row statement contains a Cell statement for each column in the table, even if a straddle hides a cell. Extra Cell statements are ignored; too few Cell statements result in empty cells in the rightmost columns of the row.

When you rotate a cell to a vertical orientation, the width of unwrapped text affects the height of the row. You can use RowMaxHeight and RowMinHeight to protect a row's height from extremes caused by rotating cells containing multiline paragraphs, or to enforce a uniform height for the rows.

FrameMaker writes out the RowHeight statement for use by other programs. It is not used by the MIF interpreter. Even if the statement is present, the MIF interpreter recalculates the height of each row based on the row contents and the RowMinHeight and RowMaxHeight statements.

Cell statement

A Cell statement specifies a cell's contents. It also includes format, straddle, and rotation information as needed. The statement must appear in a Row statement.

Syntax

<Cell	
<CellFill *integer*>	Fill pattern for cell, 0–15 (see page 113)
<CellColor *tagstring*>	Color for cell (see "ColorCatalog statement" on page 84)
<CellSeparation *integer*>	Color for cell; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
<CellLRuling *tagstring*>	Left edge ruling style (from Ruling Catalog)
<CellBRuling *tagstring*>	Bottom edge ruling style
<CellRRuling *tagstring*>	Right edge ruling style
<CellTRuling *tagstring*>	Top edge ruling style

<CellColumns *integer*>	Number of columns in a straddle cell
<CellRows *integer*>	Number of rows in a straddle cell
<CellAffectsColumnWidthA *boolean*>	Yes restricts column width to cell width
<CellAngle *degrees*>	Angle of rotation in degrees: 0, 90, 180, or 270
<CellContent	Cell's content
<Notes…>	Footnotes for cell (see page 131)
<Para…>	Cell's content, represented in one or more Para statements (see page 132)
<Para…>	Additional statements as needed
…	
>	End of CellContent statement
>	End of Cell statement

Usage

You can use the Rotate command on the Graphics menu to change the CellAngle, but it does not affect the location of cell margins. CellAngle affects only the orientation and alignment of the text flow. When CellAngle is 90 or 270 degrees, use PgfCellAlignment to move vertically oriented text closer to or farther from a column edge. For information about aligning text in a cell, see PgfCellAlignment on page 65.

MIF uses CellAffectsColumnWidthA only with the TblColumnWidthA statement. The MIF default for computing a cell's width is TblColumnWidthA. However, if any cells in the column have <CellAffectsColumnWidthA Yes>, then only those cells affect the computed column width.

Usage of MIF statements to calculate the width of a column is described in the following sections.

Determining table width

When FrameMaker writes MIF files, it uses TblColumnWidth in the Tbl statement to specify column width. However, filters that generate MIF files can use other statements to determine the table width.

This method	Uses these statements	To do this
Fixed width	TblColumnWidth	Give a fixed value for column's width (see page 77)
Shrink-wrap	TblColumnWidthA	Fit a column within minimum and maximum values (see page 77)
Restricted	TblColumnWidthA and CellAffectsColumnWidthA	Use particular cells to determine column width (see page 82)
Proportional	TblColumnWidthP	Create a temporary value for a column width when filtering proportional-width columns from another application; the MIF interpreter converts the value to a fixed width (see page 77 and "Calculating proportional-width columns," next)
Equalized	EqualizeWidths and TblColumnNum	Apply the width of the widest column to specified columns in the same table (see page 79)

The table example in "Creating an entire table" on page 238 shows several ways to determine column width.

Calculating proportional-width columns

MIF uses this formula to calculate the width of proportional-width columns:

$$\frac{n}{PTotal} \times PWidth$$

The arguments have the following values:

n	Value of `TblColumnWidthP`
PTotal	Sum of the values for all `TblColumnWidthP` statements in the table
PWidth	Available space for all proportional columns (`TblWidth` – the sum of fixed-width columns)

For example, suppose you want a four-column table to be 7 inches wide, but only the last three columns to have proportional width.

- The columns have the following widths:

  Column 1 has a fixed-width value of 1": `<TblColumnWidth 1">`

  Column 2 has a proportional value of 2: `<TblColumnWidthP 2>`

  Column 3 has a proportional value of 1: `<TblColumnWidthP 1>`

  Column 4 has a proportional value of 1: `<TblColumnWidthP 1>`
- Available width for proportional columns *(PWidth)* is 7" – 1" or 6".
- Sum of all proportional values *(PTotal)* is 2 + 1 + 1 or 4.
- Width for Column 2 is (2/*PTotal*) x *PWidth* = (2/4) x 6" or 3".
- Width for Column 3 or Column 4 is (1/*PTotal*) x *PWidth* = (1/4) x 6" or 1.5".

RulingCatalog statement

The `RulingCatalog` statement defines the contents of the Ruling Catalog, which describes ruling styles for tables. A document can have only one `RulingCatalog` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<RulingCatalog`	
`<Ruling...>`	Defines ruling style (see "Ruling statement" on page 83)
`<Ruling...>`	Additional statements as needed
...	
`>`	End of `RulingCatalog` statement

Ruling statement

The `Ruling` statement defines the ruling styles used in table formats. It must appear within the `RulingCatalog` statement.

Syntax

<Ruling	
<RulingTag *tagstring*>	Ruling style name; an empty string indicates no ruling style
<RulingPenWidth *dimension*>	Ruling line thickness
<RulingGap *dimension*>	Gap between double ruling lines
<RulingColor *tagstring*>	Color of ruling line (see "ColorCatalog statement" on page 84)
<RulingSeparation *integer*>	Color of ruling line; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
<RulingPen *integer*>	Pen pattern 0 through 7, or 15 (see page 113)
<RulingLines *integer*>	0 (none), 1 (single), or 2 (double) ruling lines
>	End of Ruling statement

Color

You can assign colors to text and objects in a FrameMaker document. A FrameMaker document has a set of default colors; you can also define your own colors and store them in the document's Color Catalog. A FrameMaker document has three color models you can use to create colors: CMYK, RGB, and HLS. You can also choose inks from installed color libraries.

In a MIF file, colors are defined by a Color statement within a ColorCatalog statement. Regardless of the color model used to define a new color, colors are stored in a MIF file in CMYK.

You can define a color as a tint of an existing color. Tints are colors that are mixed with white. A tint is expressed by the percentage of the base color that is printed or displayed. A tint of 100% is equivalent to the pure base color, and a tint of 0% is equivalent to no color at all.

You can specify overprinting for a color. However, if overprinting is set for a graphic object, the object's setting takes precedence. When a graphic object has no overprint statement, the overprint setting for the color is assumed.

You can set up color views to specify which colors are visible in a document. The color views for a document are specified in the Views statement. The current view for the document is identified in a DCurrentView statement.

The color of a FrameMaker document object is expressed in a property statement for that object. In this manual, the syntax description of a FrameMaker document object that can have a color property includes the appropriate color property substatement.

ColorCatalog statement

The ColorCatalog statement defines the contents of the Color Catalog. A document can have only one ColorCatalog statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

FrameMaker automatically generates new colors while specific operations are performed. For example, FrameMaker generates new colors when multiple conditional tags are applied to text. These colors are identified by their ColorTag, which contains the prefix "fm_gen_".

Syntax

<ColorCatalog

<Color…>	Defines a color (see "Color statement," next)
<Color…>	Additional statements as needed
…	
>	End of ColorCatalog statement

Color statement

The Color statement defines a color. It must appear within the ColorCatalog statement. Note that MIF version 5.5 and later supports multiple color libraries.

Syntax

<Color	
<ColorTag *tagstring*>	Color tag name
<ColorCyan *percentage*>	Percentage of cyan (0–100)
<ColorMagenta *percentage*>	Percentage of magenta (0–100)
<ColorYellow *percentage*>	Percentage of yellow (0–100)
<ColorBlack *percentage*>	Percentage of black (0–100)
<ColorLibraryFamilyName string>	Color library name
<ColorLibraryInkName *string*>	Specifies name of the color library pigment. The full ink name must be used.
<ColorAttribute *keyword*>	Identifies a default FrameMaker document color *keyword* can be one of: ColorIsBlack ColorIsWhite ColorIsRed ColorIsGreen ColorIsBlue ColorIsCyan ColorIsMagenta ColorIsYellow ColorIsDarkGrey ColorIsPaleGreen ColorIsForestGreen ColorIsRoyalBlue ColorIsMauve ColorIsLightSalmon ColorIsOlive ColorIsSalmon ColorIsReserved
<ColorTint percentage>	100% indicates solid color; less than 100% indicates a reduced percentage of the color
<ColorTintBaseColor string	The name of the color from which the tint is derived. If the base color does not exist in the document, black will be used.
<ColorOverprint boolean>	Yes indicates overprint is set for the color; No indicates knockout.
>	End of Color statement

Usage

In a MIF file, all colors are expressed as a mixture of cyan, magenta, yellow, and black. The `ColorAttribute` statement identifies a default FrameMaker document color; the default colors are all reserved (specified by the `ColorIsReserved` keyword) and cannot be modified or deleted by the user. A reserved default color can have two `ColorAttribute` statements, for example:

```
<ColorAttribute ColorIsCyan>
<ColorAttribute ColorIsReserved>
```

A color tint must be based on an existing color. This has two implications:

- If the base color doesn't exist in the document, black is used as the base color for the tint.
- The color value statements (values for CMYK, color family, and ink name) are ignored when included in a tint statement. However, FrameMaker writes out color value statements for a tint, even though they will be ignored. To modify the color values of a tint, modify the color value statements for the base color used by the tint.

Views statement

The `Views` statement contains the color views for the document. A document can have only one `Views` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<Views`	
`<View…>`	Defines a color view (see "View statement," next)
`<View…>`	Additional statements as needed
…	
`>`	End of `Views` statement

View statement

For each color view, the `View` statement specifies which colors will be displayed, which will be displayed as cutouts, and which will not be displayed at all. The `View` statement must appear in a `Views` statement.

Syntax

`<View`	
`<ViewNumber` *integer*`>`	View number (`1`–`6`)
`<ViewCutout` *tagstring*`>`	Name of color to print as cutout separation
`<ViewCutout…>`	Additional statements as needed
…	
`<ViewInvisible` *tagstring*`>`	Name of color to hide
`<ViewInvisible…>`	Additional statements as needed
…	
`>`	End of `View` statement

Variables

All variable definitions for a document are contained in a `VariableFormats` statement. Both user-defined and system-defined variables are defined by a `VariableFormat` statement. A `Variable` statement that refers to the variable name shows where the variable appears in text (see "ParaLine statement" on page 133).

VariableFormats and VariableFormat statements

The `VariableFormats` statement defines document variables to be used in document text flows. A MIF file can have only one `VariableFormats` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Each `VariableFormat` statement supplies a variable name and its definition. The statement must appear in a `VariableFormats` statement.

Syntax

`<VariableFormats`	
`<VariableFormat`	
`<VariableName` *tagstring*`>`	Name of variable
`<VariableDef` *string*`>`	Variable definition
`>`	End of `VariableFormat` statement
`<VariableFormat…>`	Additional statements as needed
…	
`>`	End of `VariableFormats` statement

Usage

`VariableName` contains the name of the variable, used later in the MIF file by `Variable` to position the variable in text. `VariableDef` contains the variable's definition. A system-defined variable definition consists of a sequence of building blocks, text, and character formats. A user-defined variable consists of text and character formats only.

The system variables for the current page number and running headers and footers can only appear on a master page in an untagged text flow. You cannot insert any variables in a tagged text flow on a master page. You can insert variables anywhere else in a text flow.

For more information about variables and the building blocks they can contain, see your user's manual or the online Help system.

Cross-references

A FrameMaker document can contain cross-references that refer to other portions of the document or to other documents. A cross-reference has a marker that indicates the source (where the cross-reference points) and a format that determines the text and its formatting in the cross-reference.

All cross-reference formats in a document are contained in one `XRefFormats` statement. A cross-reference format is defined by an `XRefFormat` statement. Within text, an `XRef` statement and a `Marker` statement indicate where each cross-reference appears.

XRefFormats and XRefFormat statements

The `XRefFormats` statement defines the formats of cross-references to be used in document text flows. A MIF file can have only one `XRefFormats` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

The `XRefFormat` statement supplies a cross-reference format name and its definition. The statement must appear in an `XRefFormats` statement.

Syntax

`<XRefFormats`	
`<XRefFormat`	
`<XRefName` *`string`*`>`	Cross-reference name
`<XRefDef` *`string`*`>`	Cross-reference definition
`>`	End of `XRefFormat` statement
`<XRefFormat…>`	More cross-reference definitions as needed
…	
`>`	End of `XRefFormats` statement

Usage

`XRefName` supplies the cross-reference format name, which is used later by the `XRef` statement to apply a format to the text of the cross-reference. The `XRefDef` statement supplies the cross-reference format definition, which is a string that contains text and cross-reference building blocks.

For more information about cross-references and their building blocks, see your user's manual or the online Help system.

Global document properties

A FrameMaker document has properties that specify the document page size, pagination style, view options, current user preferences, and other global document information. The user sets these properties by using various commands, such as the Document command, the View command, the Normal Page Layout command, and others.

In a MIF file, global document properties are specified as substatements in a `Document` statement. If you do not provide these property statements, the MIF interpreter assumes the properties specified in `NewTemplate`. (For information on defaults specified in templates, see page 3.)

The `BookComponent` statement specifies setup information for files generated from the document. The `Dictionary` statement contains the user's list of allowed words for the document.

Document statement

The `Document` statement defines global document properties. A document can have only one `Document` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

A `Document` statement does not need any of these property substatements, which can occur in any order. It can also contain additional substatements describing standard equation formats. (See , "MIF Equation Statements.")

Document File Info

For versions 7.0 and later, FrameMaker stores file information in packets (XMP) of encoded data. This data can be used by applications that support XMP. In MIF these data packets are expressed in the <DocFileInfo> statement. This data is generated by FrameMaker in an encoded form, and you should not edit the information. Note that this information corresponds to the values of fields in the File Info dialog box. It also corresponds to the data in the `<PDFDocInfo>` statement. However, unlike `<PDFDocInfo>`, this XMP data also includes the values of the File Info dialog box default fields for `Creator`, `Creation Date`, and `MetaData Date`.

PDF Document Info

For versions 6.0 and later, FrameMaker stores PDF File Info in the document file. FrameMaker automatically supplies values for Creator, Creation Date and Metadata Date; these Document Info fields do not appear in MIF statements for PDF Document Info. However, a user can use the File Info dialog box to specify values for Author, Title, Subject, Keywords, Copyright, Web Statement, Job Reference, and Marked. The values for all these values appear in PDF Document Info. A document can also contain arbitrary Document Info fields if they have been entered via an FDK client or by editing a MIF file. In MIF, each Document Info entry consists of one `Key` statement and at least one `Value` statement.

A `Key` statement contains a string of up to 255 ASCII characters. The `Key` names a File Info field; in PDF the field name can be up to 126 characters long. In MIF you represent non-printable characters via `#HH`, where `#` identifies a hexadecimal representation of a character, and `HH` is the hexadecimal value for the character. For example, use `#23` to represent the "#" character. Zero-value hex-codes (`#00`) are illegal. In PDF, these hexadecimal representations are interpreted as PDFDocEncoding (see *Portable Document Format Reference Manual*, Addison-Wesley, ISBN 0-201-62628-4).

Note that a a File Info field name can be up to 126 characters long, and a MIF string can contain up to 255 characters. Some characters in the key string may be hexadecimal representations, and each hexadecimal representation uses three ASCII characters. For example, a `Key` of 126 non-printing characters would require 378 ASCII characters. However, since a valid MIF string can only have up to 255 ASCII characters, such a `Key` statement would be invalid in MIF.

The contents of the File Info field is represented by a series of `Value` statements. Each value statement can contain a string of up to 255 ASCII characters. In PDF the File Info contents can contain up to 32765 Unicode characters. To accommodate this number of Unicode characters, FrameMaker generates MIF in the following ways:

- It represents the Document Info contents as a series of Value statements, each one 255 ASCII characters long, or less.
- It uses special codes to indicate Unicode characters that are outside the standard ASCII range. Mif represents Unicode characters as `&#xHHHH;`, where `&#x` opens the character code, the "`;`" character closes the character code, and `HHHH` are as many hexadecimal values as are required to represent the character.

Note that each Unicode representation of a character uses up to seven ASCII characters. For example, a string of 255 Unicode characters could require as many as 1785 ASCII characters.

For example, The following MIF statements show three possible Document Info fields:

```
<PDFDocInfo
    <Key `Author'>
    <Value `Thomas Aquinas'>
    <Key `Title'>
    <Value `That the Soul Never Thinks Without an Image'>
    <Key `Subject'>
    <Value `Modern translation of the views of T. A. concerning cognition; "It is'>
    <Value ` impossible for our intellect, in its present state of being joined t'>
    <Value `o a body capable of receiving impressions, actually to understand...'>
>          # end of PDFDocInfo
```

Syntax

<Document	Document properties
<DNextUnique *ID*>	Refers to the next object with a <Unique *ID*> statement; generated by FrameMaker and should not be used by filters
Window properties	
<DViewRect *X Y W H*>	Position and size of document window based on position and size of the document region within containing window; DViewRect takes precedence over DWindowRect
<DWindowRect *X Y W H*>	Position and size of document window based on the containing window (including the title bar, etc.)
<DViewScale *percentage*>	Current zoom setting
Column properties	
<DMargins *L T R B*>	Not generated by FrameMaker, but used by filters to specify text margins; ignored unless DColumns is specified
<DColumns *integer*>	Not generated by FrameMaker, but used by filters to specify number of columns
<DColumnGap *dimension*>	Not generated by FrameMaker, but used by filters to specify column gap
<DPageSize *W H*>	Document's default page size and orientation; if *W* is less than *H*, the document's orientation is portrait; otherwise it is landscape
Pagination	
<DStartPage *integer*>	Starting page number
<DPageNumStyle *keyword*>	Page numbering style *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha ZenLCAlpha ZenUCAlpha KanjiNumeric KanjiKazu BusinessKazu
<DPagePointStyle *keyword*>	Point page number style *keyword* can be one of: UCRoman LCRoman UCAlpha LCAlpha
<DTwoSides *boolean*>	Yes specifies two-sided layout

<DParity *keyword*>	Specifies whether first page is left or right page *keyword* can be one of: FirstLeft FirstRight
<DPageRounding *keyword*>	Method for removing blank pages or modifying total page count before saving or printing *keyword* can be one of: DeleteEmptyPages MakePageCountEven MakePageCountOdd DontChangePageCount
<DFrozenPages *boolean*>	Yes if Freeze Pagination is on
Document format properties	
<DSmartQuotesOn *boolean*>	Use curved left and right quotation marks
<DSmartSpacesOn *boolean*>	Prevents entry of multiple spaces
<DLinebreakChars *string*>	OK to break lines at these characters
<DPunctuationChars *string*>	Punctuation characters that FrameMaker does not strip from run-in heads; these characters override the default punctuation set in PgfRunInDefaultPunct (see page 64)
Conditional text defaults	
<DShowAllConditions *boolean*>	Shows or hides all conditional text
<DDisplayOverrides *boolean*>	Turns format indicators of conditional text on or off
Footnote properties	
<DFNoteTag *string*>	Paragraph and reference frame tag for document footnotes
<DFNoteMaxH *dimension*>	Maximum height allowed for document footnotes
<DFNoteRestart *keyword*>	Document footnote numbering control by page or text flow *keyword* can be one of: PerPage PerFlow
<FNoteStartNum *integer*>	First document footnote number

<DFNoteNumStyle *keyword*>	Document footnote numbering style *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha ZenLCAlpha ZenUCAlpha KanjiNumeric KanjiKazu BusinessKazu Custom
<DFNoteLabels *string*>	Characters to use in custom document footnote numbers
<DFNoteAnchorPos *keyword*>	Placement of document footnote number in text *keyword* can be one of: FNSuperscript FNBaseline FNSubscript
<DFNoteNumberPos *keyword*>	Placement of number in document footnote *keyword* can be one of: FNSuperscript FNBaseline FNSubscript
<DFNoteAnchorPrefix *string*>	Prefix before document footnote number in text
<DFNoteAnchorSuffix *string*>	Suffix after document footnote number in text
<DFNoteNumberPrefix *string*>	Prefix before number in document footnote
<DFNoteNumberSuffix *string*>	Suffix after number in document footnote
Table footnote properties	
<DTblFNoteTag *string*>	Same meaning for the following statements as the corresponding document footnote properties
<DTblFNoteLabels *string*>	
<DTblFNoteNumStyle *keyword*>	
<DTblFNoteAnchorPos *keyword*>	
<DTblFNoteNumberPos *keyword*>	
<DTblFNoteAnchorPrefix *string*>	
<DTblFNoteAnchorSuffix *string*>	
<DTblFNoteNumberPrefix *string*>	
<DTblFNoteNumberSuffix *string*>	
Change bar properties	

<DChBarGap *dimension*>	Change bar distance from column
<DChBarWidth *dimension*>	Thickness of change bar
<DChBarPosition *keyword*>	Position of change bar *keyword* can be one of: LeftOfCol RightOfCol NearestEdge FurthestEdge
<DChBarColor *tagstring*>	Change bar color (see "ColorCatalog statement" on page 84)
<DAutoChBars *boolean*>	Turns automatic change bars on or off
Document view properties	
<DGridOn *boolean*>	Turns on page grid upon opening
<DPageGrid *dimension*>	Spacing of page grid
<DSnapGrid *dimension*>	Spacing of snap grid
<DSnapRotation *degrees*>	Angle of rotation snap
<DRulersOn *boolean*>	Turns on rulers upon opening
<DFullRulers *boolean*>	Turns on formatting ruler upon opening
<DBordersOn *boolean*>	Turns on borders upon opening
<DSymbolsOn *boolean*>	Turns on text symbols upon opening
<DHotspotIndicatorsOn *boolean*>	Turns on the hotspot indicators.
<DGraphicsOff *boolean*>	Yes displays text only
<DPageScrolling *keyword*>	Specifies how FrameMaker displays consecutive pages *keyword* can be one of: Variable Horizontal Vertical Facing
<DCurrentView *integer*>	Specifies current color view (1–6)
View Only document properties	
<DViewOnly *boolean*>	Yes specifies View Only document (locked)
<DViewOnlyXRef *keyword*>	Changes behavior of active cross-references in View Only document (see page 47) *keyword* can be one of: GotoBehavior OpenBehavior NotActive

<DViewOnlySelect *keyword*>	Disables/enables user selection in View Only document, including selection with modifier keys, and sets highlighting style of destination markers for active cross-references (see "Using active cross-references" on page 47) *keyword* can be one of: No (disable user selection) Yes (enable user selection and highlighting) UserOnly (enable selection but not highlighting)
<DViewOnlyNoOp 0x*nnn*>	Disables a command in a View Only document; command is specified by hex function code (see page 48)
<DViewOnlyWinBorders *boolean*>	No suppresses display of scroll bars and border buttons in document window of View Only document
<DViewOnlyWinMenubar *boolean*>	No suppresses display of document window menu bar in View Only document
<DViewOnlyWinPopup *boolean*>	No suppresses display of document-region pop-up menus in View Only document The dotted boundary line of a document is the document-region.
<DViewOnlyWinPalette *boolean*>	Yes makes window behave as command palette window in View Only document The FrameMaker console is the Command palette window.
Document default language	
<DLanguage *keyword*>	Hyphenation and spell-checking language for text lines; for allowed keywords, see PgfLanguage on page 65
Color printing	
<DNoPrintSepColor *tagstring*>	Tag name of color not to print; any color not included here is printed. If you have multiple colors you don't want to print, use multiple statements.
<DPrintProcessColor *tagstring*>	Tag name of process color to print as separation
<DPrintSeparations *boolean*>	Yes prints separations
<DTrapwiseCompatibility *boolean*>	When printing to a PostScript file, Yes generates postscript optimized for use with the TrapWise application
<DPrintSkipBlankPages *boolean*>	Yes skips blank pages when printing
Superscripts and subscripts	
<DSuperscriptSize *percent*>	Scaling factor for superscripts expressed as percentage of the current font size
<DSubscriptSize *percent*>	Scaling factor for subscripts expressed as percentage of current font size
<DSmallCapsSize *percent*>	Scaling factor for small caps expressed as percentage of current font size
<DSuperscriptShift *percent*>	Baseline offset of superscripts expressed as percentage of current font size
<DSubscriptShift *percent*>	Baseline offset of subscripts expressed as percentage of current font size

<DSuperscriptStretch *percent*>	Amount to stretch or compress superscript, where 100% means no change
<DSubscriptStretch *percent*>	Amount to stretch or compress subscript, where 100% means no change
<DSmallCapsStretch *percent*>	Amount to stretch or compress small caps, where 100% means no change
<DRubiSize *percentage*>	The size of the rubi characters, proportional to the size of the oyamoji characters (see "Rubi text" on page 227.)
Reference properties	
<DUpdateXRefsOnOpen *boolean*>	Yes specifies that cross-references are automatically updated when the document is opened
<DUpdateTextInsetsOnOpen *boolean*>	Yes specifies that text insets are automatically updated when the document is opened
Acrobat preferences	
<DAcrobatBookmarksIncludeTagNames *boolean*>	Yes specifies that each Acrobat Bookmark title begins with the name of the paragraph tag
Document-specific menu bars	
<DMenuBar *string*>	Name of the menu bar displayed by an FDK client when the document is opened; if an empty string is specified or if the menu bar is not found, the standard FrameMaker menu bar is used
<DVoMenuBar *string*>	Name of the menu bar displayed by an FDK client when the document is opened in View Only mode; if an empty string is specified or if the menu bar is not found, the standard view-only menu bar is used
Custom catalogs	
<CustomFontFlag boolean>	Yes means the document has a custom character tag list
<CustomPgfFlag boolean>	Yes means the document has a custom paragraph formats list
<CustomTblFlag boolean>	Yes means the document has a custom table formats list
<DCustomFontList ...>	Signifies the start of the custom character tag list in the document This tag is present in the document only when you have created a custom character tag list in the document.
<DCustomFontTag *string*>	Name of the tag in the custom character tag list
<DCustomPgfList ...>	Signifies the start of the custom paragraph formats list in the document This tag is present in the document only when you have created a custom paragraph formats list in the document.
<DCustomPgfTag *string*>	Name of the paragraph tag in the custom paragraph list
<DCustomTblList ...>	Signifies the start of the custom table formats list in the document This tag is present in the document only when you have created a custom table formats list in the document.
<DCustomTblTag *string*>	Name of the table tag in the custom table tag

Math properties	For more information, see , "MIF Equation Statements."
Structure properties	For more information, see , "MIF Statements for Structured Documents and Books."
Track Text Edit properties	
<DTrackChangesOn *boolean*>	Preserves the On/Off state of the Track Text Edit option.
<DTrackChangesPreviewState *integer*>	Preserves the preview state of edited text. DTrackChangesPreviewState property can have one of the following states: • **Preview Off**: DTrackChangesPreviewState set with the value No. • **Preview On Final**: DTrackChangesPreviewState set with the value All. • **Preview On Original**: DTrackChangesPreviewState set with the value Yes.
WebDAV properties	
<WEBDAV	
<DocServerUrl *string***>**	URL of the MIF document on the WEBDAV Server. Any HTTP path is valid. Example: <DocServerUrl `http://mikej-xp/joewebdav/myfile.mif'> #http://mikej-xp/joewebdav is the path of the server.
<DocServerState>	Indicates whether the MIF document is checked in or checked out on the WebDAV server. The DocServerState property can contain one of the following values: • CheckedOut • CheckedIn
>	End of WEBDAV Document statement
Miscellaneous properties	
<DMagicMarker *integer*>	Type number of the marker used to represent a delete mark

<Document	Document properties
<DNextUnique ID>	Refers to the next object with a <Unique ID> statement; generated by FrameMaker and should not be used by filters
Window properties	
<DViewRect X Y W H>	Position and size of document window based on position and size of the document region within containing window; DViewRect takes precedence over DWindowRect
<DWindowRect *X Y W H*>	Position and size of document window based on the containing window (including the title bar, etc.)
<DViewScale percentage>	Current zoom setting

Column properties	
`<DMargins L T R B>`	Not generated by FrameMaker, but used by filters to specify text margins; ignored unless `DColumns` is specified
`<DColumns` *integer*`>`	Not generated by FrameMaker, but used by filters to specify number of columns
`<DColumnGap dimension>`	Not generated by FrameMaker, but used by filters to specify column gap
`<DPageSize W H>`	Document's default page size and orientation; if W is less than H, the document's orientation is portrait; otherwise it is landscape
Volume, chapter, and page numbering properties	
Volume numbering	
`<VolumeNumStart integer>`	Starting volume number

`<VolumeNumStyle keyword>`	Style of volume numbering
	
	keyword can be one of:
	`IndicNumeric`
	`FarsiNumeric`
	`HebrewNumeric`
	`AbjadNumeric`
	`AlifbataNumeric`
	`UCRoman`
	`LCRoman`
	`UCAlpha`
	`LCAlpha`
	`KanjiNumeric`
	`ZenArabic`
	`ZenUCAlpha`
	`ZenLCAlpha`
	`Kanjikazu`
	`BusinessKazu`
	`Custom`

`<VolumeNumText string>`	When `VolumeNumStyle` is set to `Custom`, this is the string to use
`<VolNumComputeMethod keyword>`	Volume numbering *keyword* can be one of: `StartNumbering` (restart numbering) `ContinueNumbering` (continue numbering from previous document in book) `UseSameNumbering` (use the same numbering as previous document in book)
Chapter numbering	
`<ChapterNumStart integer>`	Starting chapter number

<ChapterNumStyle keyword>	Style of chapter numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<ChapterNumText string>	When ChapterNumStyle is set to Custom, this is the string to use
<ChapterNumComputeMethod keyword>	Chapter numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous document in book) UseSameNumbering (use the same numbering as previous document in book)
Section numbering	
<SectionNumStart integer>	Starting section number
<SectionNumStyle keyword>	Style of section numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<SectionNumText string>	When SectionNumStyle is set to Custom, this is the string to use

<SectionNumComputeMethod keyword>	Section numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Sub section numbering	
<SubSectionNumStart integer>	Starting Sub section number
<SubSectionNumStyle keyword>	Style of Sub section numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<SubSectionNumText string>	When SubSectionNumStyle is set to Custom, this is the string to use
<SubSectionNumComputeMethod keyword>	Sub section numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Page numbering	

<DPageNumStyle keyword>	Page numbering style *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha ZenLCAlpha ZenUCAlpha KanjiNumeric KanjiKazu BusinessKazu
<DPagePointStyle keyword>	Point page number style *keyword* can be one of: UCRoman LCRoman UCAlpha LCAlpha
<DStartPage *integer*>	Starting page number
<ContPageNum boolean>	Yes means continue page numbering from the previous document in the book
Pagination	
<DTwoSides boolean>	Yes specifies two-sided layout
<DParity keyword>	Specifies whether first page is left or right page *keyword* can be one of: FirstLeft FirstRight
<DPageRounding keyword>	Method for removing blank pages or modifying total page count before saving or printing *keyword* can be one of: DeleteEmptyPages MakePageCountEven MakePageCountOdd DontChangePageCount
<DFrozenPages boolean>	Yes if Freeze Pagination is on
Document format properties	
<DSmartQuotesOn boolean>	Use curved left and right quotation marks
<DSmartSpacesOn boolean>	Prevents entry of multiple spaces
<DLinebreakChars string>	OK to break lines at these characters
<DPunctuationChars string>	Punctuation characters that FrameMaker does not strip from run-in heads; these characters override the default punctuation set in PgfRunInDefaultPunct (see page 64)
Conditional text defaults	
<DShowAllConditions boolean>	Shows or hides all conditional text

`<DDisplayOverrides boolean>`	Turns format indicators of conditional text on or off
Footnote properties	
`<DFNoteTag string>`	Paragraph and reference frame tag for document footnotes
`<DFNoteMaxH dimension>`	Maximum height allowed for document footnotes

`<DFNoteRestart keyword>`	Document footnote numbering control by page or text flow
	keyword can be one of:
	`PerPage`
	`PerFlow`

`<FNoteStartNum integer>`	First document footnote number

`<DFNoteNumStyle keyword>`	Document footnote numbering style
	keyword can be one of:
	`IndicNumeric`
	`FarsiNumeric`
	`HebrewNumeric`
	`AbjadNumeric`
	`AlifbataNumeric`
	`UCRoman`
	`LCRoman`
	`UCAlpha`
	`LCAlpha`
	`ZenLCAlpha`
	`ZenUCAlpha`
	`KanjiNumeric`
	`KanjiKazu`
	`BusinessKazu`
	`Custom`

`<DFNoteLabels string>`	Characters to use in custom document footnote numbers
`<DFNoteComputeMethod keyword>`	Footnote numbering *keyword* can be one of: `Continue` (continue numbering from previous component in book) `Restart` (restart numbering)
`<DFNoteAnchorPos keyword>`	Placement of document footnote number in text *keyword* can be one of: ΦΝΣυπερσχριπτ ΦΝΒασελινε ΦΝΣυβσχριπτ

`<DFNoteNumberPos keyword>`	Placement of number in document footnote
	keyword can be one of:
	`FNSuperscript`
	`FNBaseline`
	`FNSubscript`

`<DFNoteAnchorPrefix string>`	Prefix before document footnote number in text
`<DFNoteAnchorSuffix string>`	Suffix after document footnote number in text
`<DFNoteNumberPrefix string>`	Prefix before number in document footnote
`<DFNoteNumberSuffix string>`	Suffix after number in document footnote
Table footnote properties	

<DTblFNoteTag string>	Same meaning for the following statements as the corresponding document footnote properties
<DTblFNoteLabels string>	
<DTblFNoteNumStyle keyword>	
<DTblFNoteAnchorPos keyword>	
<DTblFNoteNumberPos keyword>	
<DTblFNoteAnchorPrefix string>	
<DTblFNoteAnchorSuffix string>	
<DTblFNoteNumberPrefix string>	
<DTblFNoteNumberSuffix string>	
Change bar properties	
<DChBarGap dimension>	Change bar distance from column
<DChBarWidth dimension>	Thickness of change bar
<DChBarPosition keyword>	Position of change bar *keyword* can be one of: LeftOfCol RightOfCol NearestEdge FurthestEdge
<DChBarColor tagstring>	Change bar color (see "ColorCatalog statement" on page 84)
<DAutoChBars boolean>	Turns automatic change bars on or off
Document view properties	
<DGridOn boolean>	Turns on page grid upon opening
<DPageGrid dimension>	Spacing of page grid
<DSnapGrid dimension>	Spacing of snap grid
<DSnapRotation degrees>	Angle of rotation snap
<DRulersOn boolean>	Turns on rulers upon opening
<DFullRulers boolean>	Turns on formatting ruler upon opening
<DBordersOn boolean>	Turns on borders upon opening
<DSymbolsOn boolean>	Turns on text symbols upon opening
<DGraphicsOff boolean>	Yes displays text only
<DPageScrolling keyword>	Specifies how FrameMaker displays consecutive pages *keyword* can be one of: Variable Horizontal Vertical Facing
<DCurrentView *integer*>	Specifies current color view (1–6)

View Only document properties	
`<DViewOnly boolean>`	`Yes` specifies View Only document (locked)

`<DViewOnlyXRef keyword>`	Changes behavior of active cross-references in View Only document (see page 47)
	
	keyword can be one of:
	`GotoBehavior`
	`OpenBehavior`
	`NotActive`

`<DViewOnlySelect keyword>`	Disables/enables user selection in View Only document, including selection with modifier keys, and sets highlighting style of destination markers for active cross-references (see "Using active cross-references" on page 47) *keyword* can be one of: `No` (disable user selection) `Yes` (enable user selection and highlighting) `UserOnly` (enable selection but not highlighting)
`<DViewOnlyNoOp 0xnnn>`	Disables a command in a View Only document; command is specified by hex function code (see page 48)
`<DViewOnlyWinBorders boolean>`	`No` suppresses display of scroll bars and border buttons in document window of View Only document
`<DViewOnlyWinMenubar boolean>`	`No` suppresses display of document window menu bar in View Only document
`<DViewOnlyWinPopup boolean>`	`No` suppresses display of document-region pop-up menus in View Only document The dotted boundary line of a document is the document-region.
`<DViewOnlyWinPalette boolean>`	`Yes` makes window behave as command palette window in View Only document The FrameMaker console is the Command palette window.
Document default language	
`<DLanguage keyword>`	Hyphenation and spell-checking language for text lines; for allowed keywords, see `PgfLanguage` on page 65
Color printing	
`<DNoPrintSepColor tagstring>`	Tag name of color not to print; any color not included here is printed If you have multiple colors you don't want to print, use multiple statements.
`<DPrintProcessColor tagstring>`	Tag name of process color to print as separation
`<DPrintSeparations boolean>`	`Yes` prints separations
`<DTrapwiseCompatibility boolean>`	When printing to a PostScript file, `Yes` generates postscript optimized for use with the TrapWise application
`<DPrintSkipBlankPages boolean>`	`Yes` skips blank pages when printing
Superscripts and subscripts	
`<DSuperscriptSize` *percent*`>`	Scaling factor for superscripts expressed as percentage of the current font size

<DSubscriptSize *percent*>	Scaling factor for subscripts expressed as percentage of current font size
<DSmallCapsSize *percent*>	Scaling factor for small caps expressed as percentage of current font size
<DSuperscriptShift *percent*>	Baseline offset of superscripts expressed as percentage of current font size
<DSubscriptShift *percent*>	Baseline offset of subscripts expressed as percentage of current font size
<DSuperscriptStretch *percent*>	Amount to stretch or compress superscript, where 100% means no change
<DSubscriptStretch *percent*>	Amount to stretch or compress subscript, where 100% means no change
<DSmallCapsStretch *percent*>	Amount to stretch or compress small caps, where 100% means no change
<DRubiSize percentage>	The size of the rubi characters, proportional to the size of the oyamoji characters (see "Rubi text" on page 227.)
Reference properties	
<DUpdateXRefsOnOpen *boolean*>	`Yes` specifies that cross-references are automatically updated when the document is opened
<DUpdateTextInsetsOnOpen *boolean*>	`Yes` specifies that text insets are automatically updated when the document is opened
PDF preferences	
<DAcrobatBookmarksIncludeTagNames *boolean*>	`Yes` specifies that each PDF Bookmark title begins with the name of the paragraph tag
<DPDFAllNamedDestinations boolean>	`Yes` indicates that FrameMaker will create named destinations for all paragraphs and elements in the document; this style of marking creates larger PDF files
<DPDFAllPages boolean>	A statement to indicate whether to use the values in `DPDFStart-Page` and `DPDFEndPage` to distill a range of pages. When set to `Yes`, FrameMaker distills all pages in the document.
<DPDFBookmarks boolean>	`Yes` indicates that FrameMaker will create PDF bookmarks when you save as PDF
<DPDFConvertCMYK boolean>	A setting that determines whether to send CMYK or RGB color values to the Distiller. This setting can be made and stored on documents in any platform.
<DPDFTagsConfigurationDefined boolean>	`Yes` indicates that tags configuration is defined for the PDF
<PDFTagConfigurationTOCBeginsWithList string>	A list of strings from which any TOC tag can begin
<PDFTagConfigurationTOCEndsWithList string>	A list of strings from which any TOC tag can end
<PDFTagConfigurationListBeginsWithList string>	A list of strings from which any list tag can begin
<PDFTagConfigurationListEndsWithList string>	A list of strings from which any list tag can end
<PgfPDFStructureTagType integer>	Specifies PDF tag name

<DPDFDestsMarked boolean>	Yes indicates that the paragraphs and elements that are targets of hypertext markers or cross-references have been marked according to optimization rules for version 6.0 or later; this style of marking makes it unnecessary to use <DPDFCreateNamedDestinations Yes>
<DPDFEndPage 'string'>	A string for the page number for the ending page in the page range _ to use this setting, DPDFAllPages must be set to No.
<DPDFJobOptions 'string'>	A string specifying the Distiller job options to use when distilling the document.
<DPDFOpenBookmarkLevel number>	A setting to specify at what level of the bookmark hierarchy to close all bookmarks. A setting of 0 closes all bookmarks.
<DPDFOpenFit 'string'>	A string to specify how to fit the PDF document into the Acrobat application window when it opens — can be one of Default, Page, Width, Height, or None. Any other string value resolves to Default. Use None in conjunction with DPDFOpenZoom.
<DPDFOpenPage 'string'>	A string for the page number for the page at which you want the PDF file to open.
<DPDFOpenZoom number>	A number to specify the zoom percentage when opening the PDF document. To use this setting, DPDFOpenPage must either be absent or set to None — otherwise FrameMaker ignores this setting.
<DPDFPageHeight number>	A number for the page width — to use this setting DPDFPageSizeSet must be set to Yes.
<DPDFPageSizeSet boolean>	A statement to indicate whether to use the values in DPDFPageWidth and DPDFPageHeight when distilling the document. When set to No, FrameMaker ignores the width and height settings.
<DPDFPageWidth number>	A number for the page height — to use this setting, DPDFPageSizeSet must be set to Yes
<DPDFRegMarks 'string'>	A string specifying which registration marks to use. Can be one of None, Western, or Tombo — any other string resolves to None.
<DPDFSaveSeparate Yes/No>	A setting that specifies whether to save a book as one PDF file or as a collection of separate PDF files for each component in the book. This setting is ignored in individual documents.
<DPDFStartPage 'string'>	A string for the page number for the starting page in the page range _ top use this setting, DPDFAllPages must be set to No.
<DPDFStructure boolean>	Yes indicates that the document includes structure statements for Structured PDF
<DPDFStructureDefined boolean>	Statement to determine how FrameMaker should display the PDF structure settings in the PDF Setup dialog box; this statement is for internal FrameMaker use, and you should not modify it
<PDFDocInfo>	Specifies the information that appears in the File Info dictionary when you save the document as PDF Each File Info entry consists of one Key statement followed by at least one Value statement. FrameMaker ignores any Key statement that is not followed by at least one Value statement. There is no representation in this statement of the default fields for Creator, Creation Date, or MetaData Date. For more information, see "PDF Document Info" on page 89.

<Key string>	A string of up to 255 ASCII characters that represents the name of a Document Info field; in PDF the name of a Document Info field must be 126 characters or less. Represent non-printable characters via #HH, where # identifies a hexadecimal representation of a character, and HH is the hexadecimal value for the character. For example, use #23 to represent the "#" character. Zero-value hex -codes (#00) are illegal. For more information, see "PDF Document Info" on page 89.
<Value string>	A string of up to 255 ASCII characters that represents the value of a Document Info field; because a single MIF string contains no more than 255 ASCII characters, you can use more than one `Value` statement for a given `Key` A Value can include Unicode characters; represent Unicode characters via &#xHHHH;, where &#x opens the character code, the ";" character closes the character code, and HHHH are as many hexadecimal values as are required to represent the character. For more information, see "PDF Document Info" on page 89.
...	
>	End of `PDFDocInfo` statement
<DocFileInfo>	Specifies the same information that appears in `<PDFDocInfo>`, except it expresses these values as encoded data. You should not try to edit this data. DocFileInfo also represents the values of the default fields for `Creator`, `Creation Date`, and `MetaData Date`. For more information, see "Document File Info" on page 89.
<*encoded*>	XMP information as encoded data which is generated by FrameMaker. This information corresponds to the values set in the File Info dialog box. For any document, there can be an arbitrary number of XMP statements.
...	
>	End of DocFileInfo statement
Document-specific menu bars	
<DMenuBar *string*>	Name of the menu bar displayed by an FDK client when the document is opened; if an empty string is specified or if the menu bar is not found, the standard FrameMaker menu bar is used
<DVoMenuBar *string*>	Name of the menu bar displayed by an FDK client when the document is opened in View Only mode; if an empty string is specified or if the menu bar is not found, the standard view-only menu bar is used
Math properties	For more information, see , "MIF Equation Statements."
Structure properties	For more information, see , "MIF Statements for Structured Documents and Books."
Document Direction	

<DocDir *keyword*>	Specifies the direction — left-to-right (LTR) or right-to-left (RTL), in which you can author your document. The direction of objects, which derive their direction from the document, is set to LTR or RTL. *keyword* can be one of: LTR RTL
Miscellaneous properties	
<DMagicMarker *integer*>	Type number of the marker used to represent a delete mark

BookComponent statement

BookComponent statements contain the setup information for files that are generated from the document (for example, a table of contents or an index). BookComponent statements must appear at the top level in the order given in "MIF file layout" on page 53. These statements are used even if the document does not occur as part of a book. A BookComponent statement can contain one or more DeriveTag statements.

Syntax

<BookComponent	Book components
<FileName *pathname*>	Generated file's device-independent pathname (for *pathname* syntax, see page 7)
<FileNameSuffix *string*>	Suffix for the generated file
<DeriveType *keyword*>	Type of generated file *keyword* can be one of: AML (alphabetic marker list) APL (alphabetic paragraph list) IDX (index) IOA (author index) IOM (index of markers) IOS (subject index) IR (index of references) LOF (list of figures) LOM (list of markers) LOP (list of paragraphs) LOT (list of tables) LR (list of references) TOC (table of contents) MLT (master list template)
<DeriveTag *tagstring*>	Tags to include in the generated file
<DeriveLinks *boolean*>	Yes automatically creates hypertext links in generated files
>	End of BookComponent statement

InitialAutoNums statement

The InitialAutoNums statement controls the starting values for autonumber series in a document. A MIF file can have only one InitialAutoNums statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

An autonumber format includes a series label to identify the type of autonumber series and one or more counters. The InitialAutoNums statement initializes the counters so that series that continue across files in a book are numbered correctly. Any statement that increments the counter value starts from the initial setting.

Syntax

<InitialAutoNums	
<AutoNumSeries	
<FlowTag *string*>	Specifies flow that the file uses to number the series
<Series *string*>	Specifies autonumber series
<NumCounter *integer*>	Initializes autonumber counter
<NumCounter…>	Additional statements as needed
…	
>	End of AutoNumSeries statement
<AutoNumSeries…>	Additional statements as needed
…	
>	End of InitialAutoNums statement

Dictionary statement

The Dictionary statement lists all the words in the document dictionary. A MIF file can have only one Dictionary statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

<Dictionary	
<OKWord *string*>	Word in dictionary
<OKWord *string*>	Additional statements as needed
…	
>	End of Dictionary statement

Dictionary preferences

Use the Dictionary preferences to specify Proximity or Hunspell dictionaries for Spelling and Hyphenation for various languages.

Syntax

<Dictionary	
<DiLanguages	
<DiLanguage string>	Name of the language, such as US English or Dutch.
<DiService	Name of the spelling and hyphenation service provider in the following tags under DiService: <DiSpellProvider string> <DiHyphenProvider string> You can set these tags to Hunspell or Proximity.

Pages

Pages in a MIF file are defined by a `Page` statement. A FrameMaker document can have four types of pages:

- Body pages contain the document's text and graphics.
- Master pages control the appearance of body pages.
- Reference pages contain boilerplate material or graphic art that is used repeatedly in a document, or custom math elements.
- Hidden pages contain hidden conditional text in a special text flow.

When FrameMaker writes a MIF file, it writes a sequence of numbered body pages. When you generate a MIF file, you should only define one body page and allow the MIF interpreter to automatically create new body pages as needed. For more information about using body pages in a MIF file, see "Specifying page layout" on page 32.

Page statement

The `Page` statement adds a new page to the document. `Page` statements must appear at the top level in the order given in "MIF file layout" on page 53.

Syntax

`<Page`	

`<PageType` *keyword*`>`	Page type
	
	keyword can be one of:
	`LeftMasterPage`
	`RightMasterPage`
	`OtherMasterPage`
	`ReferencePage`
	`BodyPage`
	`HiddenPage`

`<PageNum` *string*`>`	Page number for additive pages (provided for output filters)
`<PageTag` *tagstring*`>`	Names master or reference page; for a body page, specifies a different page number (for example, a point page) to be used instead of the default page number
`<PageSize` *W H*`>`	Page width and height; written by FrameMaker but ignored when a MIF file is read or imported (see `DPageSize` on page 90)
`<PageAngle` *degrees*`>`	Rotation angle of page in degrees (`0`, `90`, `180`, `270`); angles are measured in a counterclockwise direction with respect to the page's original orientation as determined by the page size (see `DPageSize` on page 90)

`<PageBackground` *keyword*`>`	Names master page to use for current page background (body pages only)
	
	keyword can be one of:
	`None`
	`Default`
	pagename

`<TextRect…>`	Defines text frame (see page 129)
`<Frame…>`	Graphic frames on the page (see the section "Graphic objects and graphic frames" on page 111)
Graphic object statements	Objects on the page (see the section "Graphic objects and graphic frames" on page 111)
Filter statements	

<HeaderL *string*>	Left header string
<HeaderC *string*>	Center header string
<HeaderR *string*>	Right header string
<FooterL *string*>	Left footer string
<FooterC *string*>	Center footer string
<FooterR *string*>	Right footer string
<HFMargins *L T R B*>	Header/footer margins
<HFFont	Header/footer font (see page 67)
<Font…>	
>	
<Columns *integer*>	Default number of columns
<ColumnGap *dimension*>	Default column gap
>	End of Page statement

Usage

Master and reference page names (supplied by the PageTag statement) appear in the status bar of a document window. The PageBackground statement names the master page to use as the background for a body page. A value of Default tells FrameMaker to use the right master page for single-sided documents and to alternate between the right and left master pages for a two-sided document. For more information about applying master page layouts to body pages, see "Specifying page layout" on page 32.

A page of type HiddenPage contains the document's hidden conditional text. (See "How FrameMaker writes a conditional document" on page 43.)

A page's size and orientation (landscape or portrait) is determined by the PageAngle statement and the Document substatement DPageSize. If DPageSize defines a portrait page (one whose height is greater than its width), pages with an angle of 0 or 180 degrees are portrait; pages with an angle of 90 or 270 degrees are landscape. If DPageSize defines a landscape page (one whose width is greater than its height), pages with an angle of 0 or 180 degrees are landscape; pages with an angle of 90 or 270 degrees are portrait.

The filter statements are not generated by FrameMaker. When it reads a MIF file generated by a filter, the MIF interpreter uses these statements to set up columns and text flows on master pages.

Mini TOC

FrameMaker document can contain a mini TOC. In a MIF file, a mini TOC tag is defined in an InlineComponentsInfo statement.

InlineComponentsInfo statement

A mini TOC is the only inline component that is available in a document. The InlineComponentsInfo statement defines the information about all type of inline components present in the document. Information about a particular type of inline component is defined using the InlineComponentInfo statement.

A MIF file can have only one `InlineComponentsInfo` statement, which must appear at the top level in the order given in the "MIF file layout" on page 53.

Syntax

`<InlineComponentsInfo`	
`<InlineComponentInfo...>`	Defines an inline component.
`> #`	End of `InlineComponentsInfo`.

InlineComponentInfo statement

The `InlineComponentInfo` statement is used to define a set of attributes with values.

Syntax

`<InlineComponentInfo`	
`<InlineComponentType MTOC>`	Type of inline component, which is the mini TOC.
`<InlineComponentLinks` *`boolean`*`>`	Specifies whether entries in an inline component are hyperlinked or not.
`<InlineComponentTag` *`string`*`>`	Name of the paragraph tags included in the inline component, for example 'Heading 1'.
`> #`	End of `InlineComponentInfo`.

Graphic objects and graphic frames

In a FrameMaker document, graphic objects can appear directly on a page or within a graphic frame. The following objects are considered graphic objects:

- Anchored and unanchored frames
- Text frames
- Text lines
- Objects created with the drawing tools on the Tools palette: arcs, arrows, ellipses, polygons, polylines, rectangles, and rounded rectangles
- Math equations
- Groups
- Imported graphic images, such as xwd, TIFF, bitmap images, or vector images

In a MIF file, graphic objects are defined by *`Object`* and `Frame` statements. *`Object`* refers to any MIF statement that describes an object, such as `Arc`, `TextLine`, or `TextRect`. Generally, these objects are created and manipulated by using the Tools palette in a FrameMaker document. This section describes general information that pertains to all graphic objects, and then lists the MIF statements for graphic objects in alphabetic order.

Object positioning

Each `Page` statement has nested within it *`Object`* and `Frame` statements. If a graphic frame contains objects and other graphic frames, the graphic frames and objects are listed in the order that they are drawn (object in back first).

For `Object` and `Frame` statements, the interpreter keeps track of the current page and current graphic frame. When the interpreter encounters a `Frame` statement, it assumes the graphic frame is on the current page. Similarly, when the interpreter encounters an object statement, it assumes the object is in the current graphic frame or page.

When you open a MIF file as a FrameMaker document, the default current page is page 1, and the default current frame is the *page frame* for page 1. A page frame is an invisible frame that "contains" objects or graphic frames placed directly on a page. The page frame is not described by any MIF statement. When you import a MIF file into an existing FrameMaker document, the default current page is the first page visible when the Import command is invoked; the current frame is the currently selected frame on that page. If there is no currently selected frame, the current frame is the page frame for that page.

Generic object statements

All object descriptions consist of the object type, generic object statements containing information that is common to all objects, and statements containing information that is specific to that type of object. This section describes the generic object statements.

Syntax

`<ID` *`ID`*`>`	Object ID number
`<GroupID` *`ID`*`>`	ID of parent group object
`<Unique` *`ID`*`>`	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
`<Pen` *`integer`*`>`	Pen pattern for lines and edges (see "Values for Pen and Fill statements" on page 113)
`<Fill` *`integer`*`>`	Fill pattern for objects (see "Values for Pen and Fill statements" on page 113)
`<PenWidth` *`dimension`*`>`	Line and edge thickness
`<ObColor` *`tagstring`*`>`	Applies color from Color Catalog (see page 84)
`<ObTint` *`percentage`*`>`	Applies a tint to the object color; 100% is equivalent to the pure object color and 0% is equivalent to no color at all
`<Separation` *`integer`*`>`	Applies color; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)
`<Overprint` *`boolean`*`>`	`Yes` turns on overprinting for the graphic object. `No` turns on knockout. If this statement is not present, then the overprint setting from the object's color is assumed.

`<RunaroundType` *`keyword`*`>`	Specifies whether text can flow around the object and, if so, whether the text follows the contour of the object or a box shape surrounding the object
	
	`keyword` can be one of:
	`Contour`
	`Box`
	`None`

`<RunaroundGap` *`dimension`*`>`	Space between the object and the text flowing around the object; must be a value between 0.0 and 432.0 points
`<Angle` *`degrees`*`>`	Rotation angle of object in degrees; default is `0` Frames, cells, and equations can only be rotated in 90-degree increments; all other objects can be arbitrarily rotated.
`<ReRotateAngle` *`dimension`*`>`	Previous rotation angle of object in degrees

<DashedPattern	
<DashedStyle *keyword*>	Specifies whether object is drawn with a dashed or a solid line *keyword* can be one of: Solid Dashed
<NumSegments *integer*>	Number of dash segments; ignored when MIF file is read
<DashSegment *dimension*>	Defines a dash segment (see "DashSegment values" on page 114)
<DashSegment *dimension*>	Additional statements as needed
...	
>	End of DashedPattern statement
<ObjectAttribute	Tagged information that gets stored with the object when you save a document as Structured PDF A graphic object can have ny number of ObjectAttribute statements
<Tag *string*>	The tag name for the object attribute
<Value *string*>	The text of the object attribute
>	End of ObjectAttribute statement

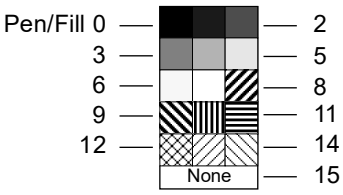
Usage

The ID substatement is necessary only if other objects refer to the object. For example, anchored frames, groups, and linked text frames require ID substatements.

The GroupID statement is necessary only if the object belongs to a set of grouped objects (Group statement). All objects in the set have the GroupID of the parent object. See "Group statement" on page 120.

Values for Pen and Fill statements

Values for the Pen and Fill statements refer to selections in the Tools palette. Graphics can use all the Pen and Fill values illustrated below. Ruling lines and table shadings use only the first seven pen/fill values and 15 (none). The pen and fill patterns might look different on your system.



Pen/Fill Patterns in Tools palette

Each Pen, Fill, or PenWidth substatement resets the MIF interpreter's corresponding current value. If an *Object* statement doesn't include one of these statements, the MIF interpreter uses the current default value for the object data.

In a FrameMaker document, patterns aren't associated directly with a document, but with FrameMaker itself. Each FrameMaker document contains indexes to FrameMaker patterns. You cannot define document patterns in MIF; you can only specify the values 0–15. However, you can customize a UNIX or Windows version of FrameMaker to use patterns that differ from the standard set. For information, see the online manuals *Customizing FrameMaker* for UNIX and *Working on Multiple Platforms* for Windows.

Values for the Angle and ReRotateAngle statements

The `Angle` statement specifies the number of degrees by which an object is rotated before it is printed or displayed. In a FrameMaker document, you can rotate an object in either a counterclockwise or clockwise direction. In a MIF file, the rotation angle is always measured in a counterclockwise direction.

An object without an `Angle` statement has an angle of 0 degrees. If an object has a `ReRotateAngle` statement, it specifies the angle to use when Esc g 0 (zero) is used to return the object to a previous rotation angle. An object with a `ReRotateAngle` statement must have an angle of 0 degrees.

The `Angle` and `ReRotateAngle` statements are mutually exclusive. When the MIF interpreter reads an `Angle` statement with a nonzero value, it sets the value of the `ReRotateAngle` statement to 0. When it reads a `ReRotate-Angle` statement with a nonzero value, it sets `Angle` to 0. Thus, if an object has both statements, the MIF interpreter keeps the state of the most recently read statement.

Objects do not inherit rotation angles from other objects.

FrameMaker rotates objects as follows:

- Polygons, polylines, and Bezier curves are rotated around the center of the edge mass.
- Text lines are rotated around the `TLOrigin` point.
- Arcs are rotated around the center of the bounding rectangle of the arc, not the bounding rectangle of the underlying ellipse. The *bounding rectangle* is the smallest rectangle that encloses an object. See your user's manual for more information about rotation.
- Other objects are rotated around the center of the object.

DashSegment values

If the `DashedStyle` statement has a value of `Dashed`, the following `DashSegment` statements describe the dashed pattern. The value of a `DashSegment` statement specifies the length of a line segment or a gap in a dashed line. See the online manual *Customizing Adobe FrameMaker* for information on changing default dashed patterns in UNIX versions of FrameMaker. In Windows versions, edit the `maker.ini` file in the directory where FrameMaker is installed. See *Customizing Adobe FrameMaker* for more information. You can also define custom dash patterns. For examples, see "Custom dashed lines" on page 236.

Values for the RunaroundType and RunaroundGap statements

The `RunaroundType` and `RunaroundGap` statements specify the styles used for the runaround properties of objects:

- If the `RunaroundType` statement is set to `Contour`, text flows around objects in the shape of the contours of the objects. The `RunaroundGap` statement specifies the distance between the objects and the text that flows around them.
- If the `RunaroundType` statement is set to `Box`, text flows around objects in the shape of boxes surrounding the objects. The `RunaroundGap` statement specifies the distance between the objects and the text that flows around them.
- If the `RunaroundType` statement is set to `None`, text doesn't flow around objects, and the value specified by the `RunaroundGap` statement is ignored.

Objects inherit the values of these statements from previous objects. Since these statements are used only to change the inherited value from a previous object, the statements are not needed for every object. For example, if you write out a MIF file, not all objects will contain these statements.

If these statements do not appear in an object or MIF file, the following rules apply:

- If an object does not contain the `RunaroundType` statement or the `RunaroundGap` statement, FrameMaker uses the values from the previous `RunaroundType` and `RunaroundGap` statements.
- If no previous `RunaroundType` and `RunaroundGap` statements exist in the MIF file, FrameMaker uses the default values `<RunaroundType None>` and `<RunaroundGap 6.0>`.
- For example, if the `<RunaroundGap 12.0>` statement appears, all objects after that statement have a 12.0 point gap from text that flows around them. If this is the only `RunaroundGap` statement in the MIF file, all objects before that statement have a 6.0 point gap (the default gap value) from the text that flows around them.
- If the MIF file does not contain any `RunaroundType` statements or `RunaroundGap` statements, FrameMaker uses the default values `<RunaroundType None>` and `<RunaroundGap 6.0>` for all objects in the file.
- For example, 3.x and 4.x MIF files do not contain any `RunaroundType` statements. When opening these files, FrameMaker uses the default value `<RunaroundType None>`, and text does not flow around any of the existing graphic objects in these files.

AFrames statement

The `AFrames` statement contains the contents of all anchored frames in a document. A document can have only one `AFrames` statement, which must appear at the top level in the order given in "MIF file layout" on page 53.

The contents of each anchored frame are defined in a `Frame` statement. Within the text flow, an `AFrame` statement indicates where each anchored frame appears by referring to the ID provided in the original frame description (see "ParaLine statement" on page 133).

Syntax

`<AFrames`	
`<Frame...>`	Defines a graphic frame (see "Frame statement" on page 117)
`<Frame...>`	Additional statements as needed
...	
`>`	End of `AFrames` statement

Arc statement

The `Arc` statement describes an arc. It can appear anywhere at the top level, or in a `Frame` or `Page` statement.

Syntax

`<Arc`	
`Generic object statements`	Information common to all objects (see page 112)

<HeadCap *keyword*>	Type of head cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<TailCap *keyword*>	Type of tail cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<ArrowStyle…>	See "ArrowStyle statement" on page 116
<ArcRect *L T W H*>	Underlying ellipse rectangle
<ArcTheta *dimension*>	Start angle
<ArcDTheta *dimension*>	Arc angle length
>	End of Arc statement

Usage

The arc is a segment of an ellipse whose bounding rectangle is defined in ArcRect. ArcTheta specifies the starting point of the arc in degrees. Zero corresponds to twelve o'clock, 90 to three o'clock, 180 to six o'clock, and 270 to nine o'clock. ArcDTheta corresponds to the length of the arc. Positive and negative values correspond to clockwise and counterclockwise extents.

ArrowStyle statement

The ArrowStyle statement defines both the head cap (at the starting point) and the tail cap (at the ending point) of lines and arcs.

The arrow style property statements can appear in any order in an ArrowStyle statement. For a complete description of arrow style properties, see your user's manual.

Syntax

<ArrowStyle	
<TipAngle *integer*>	Arrowhead tip angle in degrees
<BaseAngle *integer*>	Arrowhead base angle in degrees
<Length *dimension*>	Arrowhead length
<HeadType *keyword*>	Arrowhead type *keyword* can be one of: Stick Hollow Filled
<ScaleHead *boolean*>	Yes scales head as arrow line gets wider
<ScaleFactor *dimension*>	Scaling factor for arrowhead as line gets wider

>	End of `ArrowStyle` statement

Ellipse statement

The `Ellipse` statement describes circles and noncircular ellipses. It can appear anywhere at the top level, or in a `Frame` or `Page` statement.

Syntax

<Ellipse	
Generic object statements	Information common to all objects (see page 112)
<ShapeRect *L T W H*>	Position and size of object's bounding rectangle, before rotation, in the page or graphic frame coordinates
>	End of `Ellipse` statement

Frame statement

Usually, a `Frame` statement contains a list of *`Object`* and `Frame` statements that define the contents of the graphic frame and are listed in the draw order from back to front.

The `Frame` statement can appear at the top level or in a `Page`, `Frame`, or `AFrame` statement.

Syntax

<Frame	
Generic object statements	Information common to all objects (see page 112)
<IsHotspot *boolean*>	Whether or not the object is a hotspot.
<HotspotCmdStr *String*>	When you click on the hotspot, you can execute a command. When executed, the command takes the user to a URL or a named destination. Example syntax: 'message URL http://www.adobe.com' -Or- 'gotolink linkname'
<HotspotTitle *string*>	The tooltip text string.
<ShapeRect *L T W H*>	Position and size of object, before rotation, in page or graphic frame coordinates

<FrameType *keyword*>	Whether graphic frame is anchored, and if anchored, the position of the anchored frame *keyword* can be one of: Below Top Bottom Inline Left Right Inside Outside Near Far RunIntoParagraph NotAnchored
<AnchorDirection *keyword*>	Controls the direction of the anchored frame. *keyword* can be one of: LTR – Set the direction for the anchored frame to left to right. RTL – Set the direction for the anchored frame to right to left. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right then INHERITLTR is assigned to AnchorDirection. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left then INHERITRTL is assigned to AnchorDirection.
<Tag *tagstring*>	Name of graphic frame
<Float *boolean*>	Yes floats graphic frame to avoid large white space that results when anchored frame and the line containing it are moved to the next page
<NSOffset *dimension*>	Near-side offset
<BLOffset *dimension*>	Baseline offset
<AnchorAlign *keyword*>	Alignment of anchored frame *keyword* can be one of: Left Center Right Inside Outside
<AnchorBeside *keyword*>	Whether the graphic frame is anchored outside of a text frame or a column in a text frame *keyword* can be one of: Column TextFrame
<Cropped *boolean*>	Yes clips sides of graphic frame to fit column
<Frame…>	Other graphic frames within this frame
Graphic object statements	Objects in the graphic frame (see page 111)
>	End of Frame statement

Usage

Unless the generic object data indicates otherwise, the MIF interpreter assumes that each graphic frame inherits the properties of the current state.

A `Frame` statement that is contained within an `AFrames` statement defines an anchored frame. Any other `Frame` statement defines an unanchored frame. The assumed value for `FrameType` is `NotAnchored`.

For anchored frames, an `AFrame` statement that refers to the frame ID indicates where the anchored frame appears within the text flow (see "ParaLine statement" on page 133).

Specifications for the position and alignment of anchored frames are described in the following sections.

Position of anchored frames

The `AnchorBeside` statement determines whether the graphic frame is anchored to a text column (`Column`) or a text frame (`TextFrame`).

The `FrameType` statement specifies the position of an anchored frame. A graphic frame can be anchored within a text column or text frame or outside a text column or text frame.

If the graphic frame is anchored within a text column or text frame, the anchored frame can be positioned in one of the following ways.

If the graphic frame is anchored within a text column or text frame	The Frame statement contains
At the insertion point of the cursor	`<FrameType Inline>`
At the top of the text column	`<FrameType Top>`
Below the insertion point of the cursor	`<FrameType Below>`
At the bottom of the text column	`<FrameType Bottom>`
Running into the paragraph	`<FrameType RunIntoParagraph>`

If the graphic frame is anchored outside a text column or a text frame, the anchored frame can be positioned in one of the following ways.

If the graphic frame is anchored outside a text column or text frame	The Frame statement contains
On the left side of the text column or text frame	`<FrameType Left>`
On the right side of the text column or text frame	`<FrameType Right>`
On the side of the text column or text frame closer to the binding of the book (the "inside edge")	`<FrameType Inside>`
On the side of the text column or text frame farther from the binding of the book (the "outside edge")	`<FrameType Outside>`
On the side of the text column or text frame closer to any page edge	`<FrameType Near>`
On the side of the text column or text frame farther from any page edge	`<FrameType Far>`

Alignment of anchored frames

If a graphic frame is anchored within a text column or text frame, the `AnchorAlign` statement specifies the alignment of the anchored frame. Unless anchored at the insertion point of the cursor, the graphic frame can be aligned in one of the following ways.

If the graphic frame is aligned	The Frame statement contains
With the left side of the text column or text frame	`<AnchorAlign Left>`
In the center of the text column or text frame	`<AnchorAlign Center>`
With the right side of the text column or text frame	`<AnchorAlign Right>`

If the graphic frame is aligned	The Frame statement contains
With the side of the text column or text frame closer to the binding of the book (the "inside edge")	`<AnchorAlign Inside>`
With the side of the text column or text frame farther from the binding of the book (the "outside edge")	`<AnchorAlign Outside>`

Group statement

The `Group` statement defines a group of graphic objects and allows objects to be nested. The `Group` statement must appear after all the objects that form the group. It can appear at the top level or within a `Page` or `Frame` statement.

Syntax

`<Group`	
`<ID` *`ID`*`>`	Group ID
`<Unique` *`ID`*`>`	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
`<Angle…>`	Rotation angle of group (see page 112)
`>`	End of `Group` statement

Usage

When the MIF interpreter encounters a `Group` statement, it searches all objects within the current graphic frame for those group IDs that match the ID of the `Group` statement. These objects are then collected to form the group. All objects with the same group ID must be listed in the MIF file before their associated `Group` statement is listed. If multiple `Group` statements have the same ID, the results will be unpredictable. For more information about the group ID, see "Generic object statements" on page 112.

ImportObject statement

The `ImportObject` statement describes an imported graphic. It can appear at the top level or within a `Page` or `Frame` statement.

The imported graphic is either copied into the document or imported by reference:

- If the imported graphic is copied into the document, the data describing the graphic is recorded within the `ImportObject` statement. The description of a graphic in a given format is called a *facet*.
- FrameMaker uses facets to display graphics, print graphics, or store additional graphic information. Imported graphics can have more than one facet, which means that the graphic is described in more than one format.
- If the graphic is imported by reference, the data describing the graphic is not stored within the `ImportObject` statement. Instead, a directory path to the file containing the graphic data is recorded in the `ImportObject` statement.

Syntax

`<ImportObject`	
`Generic object statements`	Information common to all objects (see page 112)
`<ImportObFile` *`pathname`*`>`	Object's UNIX-style pathname; no longer used, but written out by FrameMaker for backward-compatibility

<ImportObFileDI *pathname*>	Object's device-independent pathname (see page 7)
<ImportHint *string*>	Record identifying the filter used for graphics imported by reference (see "Record of the filter used to import graphic by reference" on page 124)
<PosterFileDI *pathname*>	Poster file's pathname A poster file is the default image that appears when the movie is not playing. By default, either standard icons or the first frame of the movie is used as its poster image.
<ShapeRect *L T W H*>	Position and size of object, before rotation, in the page or graphic frame coordinates
<BitMapDpi *integer*>	Scaling value for bitmap file; ignored for FrameVector graphics
<ImportObFixedSize *boolean*>	`Yes` inhibits scaling of bitmap file (see "Size, position, and angle of imported graphics" on page 122); ignored for FrameVector graphics
<opacity integer>	Opacity value defined in an object style
<FlipLR *boolean*>	`Yes` flips object about the vertical axis
<ImportObNameDI *pathname*>	This contains a name for the inset object but it is valid only of inset contains facets FLV, SWF and U3D.
<ObjectActivateInPDF *boolean*>	On creation of PDF, if this flag is ON for the object, the corresponding annotation in PDF will get active as soon as the page containing this object becomes visible. This is only valid for inset having facets FLV, SWF and U3D.
<ObjectOpenInFloatWindow *boolean*>	On creation of PDF, if this flag is ON for the object, the corresponding annotation in PDF will open in new window inside PDF reader soon as the page containing this object becomes visible. This is only valid for inset having facets FLV, SWF and U3D.
<ObjectSupportMMLink *boolean*>	This tag represents if the inset support creation of multimedia link to it from text. The inset having facets FLV, SWF and U3D supports this.
=*string*	Specifies the name of the facet used to describe the graphic imported by copying (see , "Facet Formats for Graphics.")
&%*keyword*	Identifies the data type used in the facet (see , "Facet Formats for Graphics."). *keyword* can be one of: `v` (for unsigned bytes) `i` (for integer data) `m` (for metric data)
&...	Data describing the imported graphic; data must begin with the ampersand character (see , "Facet Formats for Graphics.")
&\x	Marks the beginning or end of data represented in hexadecimal (see , "Facet Formats for Graphics.")
=EndInset	End of the data describing the imported graphic
<NativeOrigin *X Y*>	Coordinates of the origin of the imported graphic within the page or frame; applicable for graphics that use coordinate systems, such as EPS
<ImportObEditor *string*>	Name of application to call to edit bitmap graphic inset or imported object; ignored for FrameVector graphics
<ImportObUpdater *string*>	Identifies the imported graphic object or an embedded Windows OLE object. For a description of the syntax of the string, see "Methods of importing graphics" on page 123.

`<ImportURL string>`	The http file path of graphic files imported by reference
`<ObjectInfo string>`	U3D model properties such as lighting scheme, background color, existing view, and rendering mode. The properties specified in this tag are applied to the U3D object when a MIF file containing a U3D object is opened in FrameMaker. **Description of record**: `<view name>;<color>;<lighting scheme>;<rendering mode>` • `<view name>`: Valid view of the given U3D object • `<lighting scheme>`: Valid values are from "-2" to "9" (where "-2" corresponds to 'Lights From File' and "9" to 'HeadLamp') • `<rendering mode>`: Valid values are from "1" to "15" (where "1" corresponds to 'Bounding Box' and "15" to 'Hidden Wireframe') **Example**: `<ObjectInfo `camera1;16777215;6;8;'>`
>	End of `ImportObject` statement

Usage

The `ImportObject` statement describes the imported graphic's position, size, and angle. If the graphic is imported by reference, the statement describes the path to the graphic file. If the imported graphic is copied into the document, the statement contains the data describing the graphic. Data describing the graphic is stored in one or more facets. If the graphic is linked with an application (through FrameServer or an FDK client), the statement also describes the path to the application used to edit the graphic.

Usage of some of the aspects of the `ImportObject` statement is described in the following sections.

Graphic file formats

You can import different types of graphic files into a FrameMaker document.

Bitmaps: The term *bitmap graphics* (also called raster graphics) refers to graphics represented by bitmap data. Graphics file formats recognized by FrameMaker include FrameImage, Sun™ rasterfile, xwd, TIFF, PCX, and GIF files.

Vector: The term *vector graphics* (also called object-oriented graphics) refers to graphics represented by geometric data. Graphics file formats recognized by FrameMaker include FrameVector, CGM, Corel Draw, Micrografx Drawing Format, DXF, EPS, GEM, HPGL, IGES, PICT, WMF, and WPG. Note that some of these graphic file formats can also contain bitmap data.

Size, position, and angle of imported graphics

When you import a MIF file, FrameMaker determines the size of the graphic by the graphic type and the value of the `ImportObFixedSize` statement.

If the file format is	Image scaled	Size determined by
Bitmap with `<ImportObFixedSize Yes>`	No	`ShapeRect` statement
Bitmap with `<ImportObFixedSize No>`	Yes	`BitMapDpi` statement
Vector	Yes	Dimensions specified in the vector data
Encapsulated PostScript, QuickDraw PICT	No	Bounding box information in imported image

Position and coordinate systems: Some types of graphics (such as EPS) use coordinate systems to specify the position of the graphic. When these types of graphics are imported into a FrameMaker document, the `NativeOrigin` statement specifies the coordinates of the origin of the graphic within the page or frame. If the imported graphic is updated, FrameMaker uses the coordinates from the `NativeOrigin` statement to prevent the graphic from shifting on the page or frame.

Size and scale of TIFF graphics: FrameMaker doesn't use internal TIFF dpi information for sizing purposes because not all TIFF files contain that information and because it may be incorrect. FrameMaker allows users to set the dpi manually when importing the TIFF file. Once the graphic is imported, FrameMaker displays the dpi information in the Object Properties dialog box.

Angle of imported graphics: If an object contains both a `<FlipLR Yes>` statement and an `Angle` statement with a nonzero value, the object is first flipped around the vertical axis and then rotated by the value specified in `Angle`.

Methods of importing graphics

As mentioned previously, an imported graphic can be imported by reference or copied into the document. In the Windows version, an imported graphic can be a SWF object.

The following table shows how the structure of the `ImportObject` statement differs, depending on how the graphic is imported. For an explanation of the facet syntax, see , "Facet Formats for Graphics."

If the graphic is	The ImportObject statement contains
Copied into the FrameMaker document	`=`*`facet_name`* `&`*`data_type`* `&`*`facet_data`* `=EndInset`

Imported by reference	`<ImportObFileDI` *`pathname`*`>`
	
	`<ImportHint` *`string`*`>`

Imported graphic or embedded OLE object (Windows only)	*`= facet_name`* `of an imported graphic object or an OLE object` *`&data_type`* *`&facet_data`* *`=facet_name`* *`&data_type`* *`&facet_data`* `=EndInset` `Example: <ImportObUpdater` `` `SWF' ``>

Filenames of objects imported by reference

When an object is imported by reference to an external file, the `ImportObject` statement contains the file pathname.

The `ImportObFileDI` statement specifies the pathname for graphics imported by reference. The statement supplies a device-independent pathname so that files can easily be transported across different types of systems (see "Device-independent pathnames" on page 7).

In previous versions of FrameMaker, the `ImportObFile` statement was used to specify the pathname for graphics imported by reference. The statement, which is no longer used, supplies a UNIX-style pathname, which uses a slash (/) to separate directories (for example, `<ImportObFile` `` `/usr/doc/template.mif' ``>). FrameMaker still writes the `ImportObFile` statements to a MIF file for compatibility with version 1.0 of FrameMaker.

Facets in imported graphics

If a graphic is copied into a document, the data describing the graphic is stored as facets in the MIF file. (Graphics imported by reference also use facets, but these are temporary and are not saved to the file. A MIF file with a graphic imported by reference does not contain any facets.)

A facet contains graphic data in a specific format. For example, a TIFF facet contains graphic data described in TIFF format. An EPSI facet contains graphic data in EPSI format.

Facets and facet formats are described in the appendixes of this manual:

- For a general description of facets and facet formats, see , "Facet Formats for Graphics."
- For a description of the facet format for EPSI graphic data, see , "EPSI Facet Format."
- For a description of the FrameImage format used in facets, see , "FrameImage Facet Format."
- For a description of the FrameVector format in facets, see , "FrameVector Facet Format."

Record of the filter used to import graphic by reference

The `ImportHint` statement contains a record to identify the filter that was used to import the graphic by reference. FrameMaker uses the record to find the correct filter to reimport the graphic when a user opens the document again.

Note that for graphics imported by copy, FrameMaker uses the facet name stored with the graphic. The `ImportHint` statement is not written for graphics imported by copy.

The record specified by the `ImportHint` statement uses the following syntax:

`record_vers vendor format_id platform filter_vers filter_name`

Note that the fields in the record are not separated by spaces. For example:

```
`0001PGRFPICTMAC61.0 Built-in PICT reader'
```

The rest of this section describes each field in the record.

`record_vers` is the version on the record (for example, 0001).

`vendor` is a code specifying the filter's vendor. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
`'PGRF'`	Built-in FrameMaker filters
`'FAPI'`	External FDK client filter
`'FFLT'`	External FrameMaker filters
`'IMAG'`	External ImageMark filters
`'XTND'`	External XTND filters

Note that this is not a comprehensive list of codes. Codes may be added to this list by Adobe or by developers at your site.

`format_id` is a code specifying the format that the filter translates. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
`'PICT'`	QuickDraw PICT
`'WMF'`	Windows MetaFile
`'EPSF'`	Encapsulated PostScript (Macintosh)

Code	Description
'EPSI'	Encapsulated PostScript Interchange
'EPSB'	Encapsulated PostScript Binary (Windows)
'EPSD'	Encapsulated PostScript with Desktop Control Separations (DCS)
'SNRF'	Sun Raster File
`PNTG'	MacPaint
'PCX'	PC Paintbrush
'TIFF'	Tag Image File Format
'XWD'	X Windows System Window Dump file
'GIF'	Graphics Interchange Format (CompuServe)
'MIF'	Maker Interchange Format
'FRMI'	FrameImage
'FRMV'	FrameVector
'SRGB'	SGI RGB
'CDR'	CorelDRAW
'CGM'	Computer Graphics Metafile
'DRW'	Micrografx CAD
'DXF'	Autodesk Drawing eXchange file (CAD files)
'GEM'	GEM file (Windows)
'HPGL'	Hewlett-Packard Graphics Language
'IGES'	Initial Graphics Exchange Specification (CAD files)
'WPG'	WordPerfect Graphics
'DIB'	Device-independent bitmap (Windows)
'OLE'	Object Linking and Embedding Client (Microsoft)
'EMF'	Enhanced MetaFile (Windows)
'MooV'	QuickTime Movie
'IMG4'	Image to CCITT Group 4 (UNIX)
'G4IM'	CCITT Group 4 to Image
'SWF'	Shockwave Flash file
'U3D'	U3D file format

Note that this is not a comprehensive list of codes. Codes may be added to this list by Adobe or by developers at your site.

platform is a code specifying the platform on which the filter was run. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
'WINT'	Windows NT®
'WIN3'	Windows 3.1
'WIN4'	Windows 95
'UNIX'	Generic X/11 (Sun, HP)

filter_vers is a string of four characters identifying the version of the filter on that platform. For example, version 1.0 of a filter is represented by the string `1.0 '.

filter_name is a text string (less than 31 characters long) that describes the filter.

Importing a Flash file

When a Flash file is imported into a FrameMaker document, the filter_id data is rendered as a device independent bitmap (DIB). You can import a Shockwave Flash (SWF) file by referencing it from the document or by pasting it into the document. In both cases, the graphic object is made up of two facets—DIB and SWF—that are streamed when the document is saved as a MIF file.

Importing a U3D file

When a U3D file is imported into a FrameMaker document, the filter_id data is rendered as a device independent bitmap (DIB).You can import a U3D file by referencing it from the document or by pasting it into the document. In both cases, the graphic object is made up of two facets—DIB and U3D—that are streamed when the document is saved as a MIF file. When you import a U3D file by reference, the MIF file contains the name and path of the U3D file.

More information about imported graphics

For additional information on imported graphics, consult one of the following sources:

- For instructions about modifying an application to create graphic insets for FrameMaker documents, see the *FDK Programmer's Guide*.
- If you are using FrameServer or Live Links with graphic insets, see the online manual, *Using FrameServer with Applications and Insets*, which is included in the UNIX version of the Frame Developer's Kit.
- For more information about importing graphics, see your user's manual.

Math statement

A `Math` statement describes an equation. For its description, see , "MIF Equation Statements."

Polygon statement

The `Polygon` statement describes a polygon. It can appear at the top level or in a `Page` or `Frame` statement.

Syntax

<Polygon	
Generic object statements	Information common to all objects (see page 112)

<Smoothed *boolean*>	Yes smooths angles to rounded curves
<NumPoints *integer*>	Number of vertices
<Point *X Y*>	Position of object in page or frame coordinates
...	More points as needed
>	End of Polygon statement

Usage

The NumPoints statement is optional. When the MIF interpreter reads a MIF file, it counts the Point statements to determine the number of points in the polygon.

PolyLine statement

The PolyLine statement describes a polyline. It can appear at the top level or in a Page or Frame statement.

Syntax

<PolyLine	
Generic object statements	Information common to all objects (see page 112)
<HeadCap *keyword*>	Type of head cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<TailCap *keyword*>	Type of tail cap for lines and arcs *keyword* can be one of: ArrowHead Butt Round Square
<ArrowStyle...>	See "ArrowStyle statement" on page 116
<Smoothed *boolean*>	Yes smooths angles to rounded curves
<NumPoints *integer*>	Number of vertices
<Point *X Y*>	Position in page or graphic frame coordinates
...	More points as needed
>	End of PolyLine statement

Usage

The PolyLine statement is used for both simple and complex lines. A simple line is represented as a PolyLine with <NumPoints 2>. The NumPoints statement is optional. When the MIF interpreter reads a MIF file, it counts the Point statements to determine the number of points in the polyline.

Rectangle statement

The `Rectangle` statement describes rectangles and squares. It can appear at the top level or in a `Page` or `Frame` statement.

Syntax

`<Rectangle`	
`Generic object statements`	Information common to all objects (see page 112)
`<ShapeRect` *`L T W H`*`>`	Position and size of object, before rotation, in page or graphic frame coordinates
`<Smoothed` *`boolean`*`>`	`Yes` smooths angles to rounded curves
`>`	End of `Rectangle` statement

RoundRect statement

A `RoundRect` statement describes a rectangle with curved corners. It can appear at the top level or in a `Page` or `Frame` statement.

Syntax

`<RoundRect`	
`Generic object statements`	Information common to all objects (see page 112)
`<ShapeRect` *`L T W H`*`>`	Position and size of object, before rotation, in page or graphic frame coordinates
`<Radius` *`dimension`*`>`	Radius of corner; 0=square corner
`>`	End of `RoundRect` statement

TextLine statement

The `TextLine` statement describes a text line. It can appear at the top level or in a `Page` or `Frame` statement.

A text line is a single line of text that FrameMaker treats differently from other text. Text lines grow and shrink as they are edited, but they do not automatically wrap the way text in a text column does. Text lines cannot contain paragraph formats, markers, variables, cross-references, or elements.

Syntax

`<TextLine`	
`Generic object statements`	Information common to all objects (see page 112)
`<TLOrigin` *`X Y`*`>`	Alignment point origin

`<TLAlignment` *`keyword`*`>`	Alignment
	`keyword` can be one of:
	`Center`
	`Left`
	`Right`

<TLDirection *keyword*>	Controls the direction in which the text line is drawn. *keyword* can be one of: LTR – Set the direction for the text line object to left to right. RTL – Set the direction for the text line object to right to left. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right then INHERITLTR is assigned to TLDirection. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left then INHERITRTL is assigned to TLDirection.
<TLLanguage *keyword*>	Spell checking and hyphenation language for text line; for list of allowed keywords, see PgfLanguage on page 65
<Char *integer*>	Nonprinting ASCII character code
<Font…>	Embedded font change (see "PgfFont and Font statements" on page 67)
<String *string*>	Printable ASCII text in single quotation marks; required
>	End of TextLine statement

Usage

The TLOrigin statement specifies the baseline (Y) and the left, center, or right edge of the text line (X), depending on TLAlignment. The text line is rotated by the value specified in an Angle statement. The default angle is 0.

A TextLine statement contains one or more String statements. Each String statement is preceded by an optional Font statement. The Char statements provide codes for characters outside the printable ASCII range. You can define macros that make Char statements more readable, and there are several predefined constants for character values. (See "Char statement" on page 134.)

TextRect statement

The TextRect statement defines a text frame. It can appear at the top level or in a Page or Frame statement.

Syntax

<TextRect	
Generic object statements	Information common to all objects (see page 112)
<ShapeRect *L T W H*>	Position and size of object, before rotation, in page or graphic frame coordinates
<TRNext *integer*>	ID of next text frame in flow
<TRNumColumns *integer*>	Number of columns in the text frame (1–10)
<TRColumnGap *dimension*>	Space between columns in the text frame (0"–50")
<TRColumnBalance *boolean*>	Yes means columns in the text frame are automatically adjusted to the same height
<TRSideheadWidth *dimension*>	Width of side head area (0"–50")
<TRSideheadGap *dimension*>	Gap between side head area and body text area (0"–50")

<TRSideheadPlacement *keyword*>	Placement of side head in text frame *keyword* can be one of: Left Right Inside Outside
<TextFlow	See "Text flows," next
>	End of TextRect statement

Usage

A text frame can contain one or more text columns (up to ten text columns). The number of columns and the space between columns are specified by the TRNumColumns and TRColumnGap statements, respectively. The space between columns cannot exceed 50 inches.

FrameMaker can adjust the height of the text columns to evenly distribute the text in the columns if the TRColumn-Balance statement is set to Yes.

A text frame also contains the specifications for the placement of side heads. The width and location of the side head in a text frame are specified by the TRSideheadWidth and TRSideheadPlacement statements. The side head area cannot be wider than 50 inches. In the TRSideheadPlacement statement, the Inside and Outside settings correspond to the side closer to the binding and the side farther from the binding, respectively. The spacing between the side head and the text columns in the text frame is specified by the TRSideheadGap statement. The spacing cannot exceed 50 inches.

TRNext indicates the ID of the next text frame in the flow. If there is no next TextRect, use a <TRNext 0> statement or omit the entire TRNext statement. The text frame is rotated by the value specified in an Angle statement. The default angle is 0.

Text flows

Text flows contain the actual text of a FrameMaker document. In a MIF file, text flows are contained in TextFlow statements. Typically, the TextFlow statement consists of a list of embedded Para statements that contain paragraphs, special characters, table and graphic frame anchors, and graphic objects.

When the MIF interpreter encounters the first TextFlow statement, it sets up a default text flow environment. The default environment consists of the current text frame, current paragraph properties, and current font properties. The TextFlow statement can override all of these defaults.

TextFlow statement

The TextFlow statement defines a text flow. It can appear at the top level or in a TextRect statement. It must appear after all other main statements in the file.

Syntax

<TextFlow	
<TFTag *tagstring*>	Text flow tag name
<TFAutoConnect *boolean*>	Yes adds text frames as needed to extend flows

<FlowDir *keyword*>	Controls the flow direction and of the direction of child objects that derive their direction from the flow. *keyword* can be one of: LTR – Set the direction of the text flow object to left to right. The text flow propagates its direction to all child objects that derive their direction from the text flow object. RTL – Set the direction of the text flow object to right to left. The text flow propagates its direction to all child objects that derive their direction from the text flow object. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right, then INHERITLTR is assigned to FlowDir. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left, then INHERITRTL is assigned to FlowDir.
<TFPostScript *boolean*>	Yes identifies text in the flow as printer code
<TFFeather *boolean*>	Yes adjusts vertical space in column so that last line of text lies against the bottom of the column
<TFSynchronized *boolean*>	Yes aligns baselines of text in adjacent columns
<TFLineSpacing *dimension*>	Line spacing for synchronized baselines
<TFMinHangHeight *dimension*>	Maximum character height for synchronization of first line in column; if characters exceed this height, FrameMaker doesn't synchronize the first line
<TFSideheads *boolean*>	Yes means text flow contains side heads
<TFMaxInterLine *dimension*>	Maximum interline spacing
<TFMaxInterPgf *dimension*>	Maximum interparagraph spacing
<Notes…>	Defines a footnote (see "Notes statement," next)
<Para…>	Defines a paragraph (see "Para statement" on page 132)
>	End of TextFlow statement

Usage

Most MIF generators will put all document text in one TextFlow statement. However, if there are subsequent TextFlow statements, the interpreter assumes they have the same settings (current paragraph format, current font, and so forth) as the previous text flow.

To divert the flow into a new, unlinked text frame, there must be a TextRectID statement in the first ParaLine statement of the new TextFlow statement (see page 133). The TextRectID statement resets the current text frame definition so subsequent text is placed within the identified text frame; this is necessary only if you want to reset the text frame defaults.

If the text flow contains side heads, the TFSideheads statement is set to Yes. The PgfPlacementStyle statement (under paragraph properties) identifies the side heads, and the TextRect statement contains specifications for their size and placement.

For information about text flow properties, see your user's manual.

Notes statement

The Notes statement defines all of the footnotes that will be used in a table title, cell, or text flow. It can appear at the top level or at the beginning of a TblTitleContent, CellContent, or TextFlow statement.

Syntax

<Notes	
<FNote	
<ID *ID*>	Unique ID
<Unique *ID*>	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
<Font…>	Changes font as needed (see "PgfFont and Font statements" on page 67)
<Para…>	Footnote text (see "Para statement," next)
<Para…>	Additional statements as needed
>	End of FNote statement
<FNote…>	Additional statements as needed
>	End of Notes statement

Usage

Within the document text, footnotes are referred to with the <FNote *ID*> statement, where *ID* is the ID specified in the corresponding FNote statement. See "ParaLine statement" on page 133.

Para statement

The Para statement defines a paragraph. It can appear in a TextFlow, FNote, CellContent, or TblTitleContent statement. In simple MIF files without page or document statements (such as the hello.mif sample file), the Para statement can also appear at the top level. It usually consists of a list of embedded ParaLine statements that contain the document text.

Syntax

<Para	
<Unique *ID*>	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
<PgfTag *tagstring*>	Applies format from Paragraph Catalog
<Pgf…>	Sets current paragraph format (see page 62)
<PgfNumString *string*>	Paragraph number (contains the actual string)
<PgfEndCond *boolean*>	Used only for hidden conditional text; Yes indicates this is the last paragraph in the current block of conditional text in the HIDDEN text flow (see page 43)
<PgfCondFullPgf *boolean*>	Used only for hidden conditional text; Yes indicates paragraph contains end of current block of hidden text and current block ends with a paragraph symbol
<ParaLine…>	See "ParaLine statement," next
>	End of Para statement

Usage

By default, a paragraph uses the current `Pgf` settings (the same settings as its predecessor). Optional `PgfTag` and `Pgf` statements reset the current format. If there is a `PgfTag` statement, the MIF interpreter searches the document's Paragraph Catalog for a `Pgf` definition with the same tag. If the tag exists, then the Paragraph Catalog's `Pgf` definition is used. If no definition is found in the catalog, the `Pgf` definition of the previous paragraph is used; however, its tag string is reset to the tag in the `PgfTag` statement.

ParaLine statement

The `ParaLine` statement defines a line within a paragraph. It must appear in a `Para` statement.

Syntax

<ParaLine	
<ElementBegin …>	See , "MIF Statements for Structured Documents and Books."
<TextRectID *ID*>	Where the following text goes
<InlineComponent	
<Unique *ID*>	Unique ID number assigned by FrameMaker.
<InlineComponentType MTOC>	Type of inline component, which is the mini TOC.
> #	End of `InlineComponent` statement.
<InlineComponentEnd >	End of inline component content.
<SpclHyphenation *boolean*>	Hyphenation of a word at the end of a line causes the word to be spelled differently, as with German hyphenation
<Font…>	Embedded character change for the following text (see page 67)
<Conditional…>	Turns on conditional text (see page 58)
<Unconditional>	Returns to unconditional state
<String *string*>	Printable ASCII text in single quotation marks; required
<Char…>	An extended ASCII character code or special character name (see page 134)
<ATbl *ID*>	ID of embedded table
<AFrame *ID*>	ID of embedded anchored frame
<FNote *ID*>	ID of embedded footnote
<Marker…>	Embedded marker (see page 135)
<Variable	Embedded variable
<VariableName *string*>	Variable name (see page 87)
<VariableLocked *boolean*>	`Yes` means the variable is part of a text inset that obtains its formatting information from the source document
>	End of `Variable` statement
<XRef…>	Embedded cross-reference (see page 88)

<XRefEnd>	
<ElementEnd ...>	See , "MIF Statements for Structured Documents and Books."
>	End of ParaLine statement

Usage

A typical ParaLine statement consists of one or more String, Char, ATbl, AFrame, FNote, Variable, XRef, and Marker statements that define the contents of the line of text. These statements are interspersed with statements that indicate the scope of document components such as structure elements and conditional text.

The VariableLocked statement is used for text insets that retain formatting information from the source document.

If the <VariableLocked Yes> statement appears in a specific variable, that variable is part of a text inset that retains formatting information from the source document. The variable is not affected by global formatting performed on the document.

If the <VariableLocked No> statement appears in a specific variable, that variable is not part of a text inset or is part of a text inset that reads formatting information from the current document. The variable is affected by global formatting performed on the document.

For more information about text insets, see "Text insets (text imported by reference)" on page 138.

Char statement

The Char statement inserts an extended ASCII character in a ParaLine statement. It must appear in a ParaLine, TextLine, or BookXRef statement.

Syntax

<Char *keyword*>	Preset name for special character (for allowed *keyword* values, see "Usage," next)

Usage

To include an extended ASCII character in a ParaLine statement, use the Char statement with a predefined character name.

For example, you can represent the pound sterling character (£) with the statement <Char Pound>, as shown in the following example:

```
<Para
 <ParaLine
  <String `the pound sterling'>
  <Char Pound>
  <String ` symbol'>
 >              # end of ParaLine
>               # end of Para
<Para
 <ParaLine
  <String `the pound sterling \xa3 symbol'>
 >              # end of ParaLine
>               # end of Para
```

You can use the <Char HardReturn> statement to insert a forced return in a paragraph. The <Char HardReturn> statement must be the last substatement in a ParaLine statement.

```
<Para
 <ParaLine
  <String `string 1'>
  <Char HardReturn>
```

```
 >              # end of ParaLine
 <ParaLine
  <String `string 2'>
 >              # end of ParaLine
>               # end of Para
```

For a list of character codes, see the *Quick Reference* for your FrameMaker product. Use the `Char` statement for a small set of predefined special characters.

Character name	Description
`Tab`	Tab
`HardSpace`	Nonbreaking space
`SoftHyphen`	Soft hyphen
`HardHyphen`	Nonbreaking hyphen
`DiscHyphen`	Discretionary hyphen
`NoHyphen`	Suppress hyphenation
`Cent`	Cent (¢)
`Pound`	Sterling (£)
`Yen`	Yen (¥)
`EnDash`	En dash (–)
`EmDash`	Em dash (—)
`Dagger`	Dagger (†)
`DoubleDagger`	Double dagger (‡)
`Bullet`	Bullet (•)
`HardReturn`	Forced return
`NumberSpace`	Numeric space
`ThinSpace`	Thin space
`EnSpace`	En space
`EmSpace`	Em space

In MIF 8 documents, the following 10 special characters are no longer represented by Character Names. You can directly enter the UTF-8 code points of these characters:

- `<Char DiscHyphen>`
- `<Char NoHyphen>`
- `<Char HardHyphen>`
- `<Char Tab>`
- `<Char HardReturn>`
- `<Char NumberSpace>`
- `<Char HardSpace>`
- `<Char ThinSpace>`
- `<Char EnSpace>`
- `<Char EmSpace>`

However, these special characters continue to be represented by Character Names in dialog boxes.

MarkerTypeCatalog statement

The `MarkerTypeCatalog` statement defines the contents of the catalog of user-defined markers for the current document. A document can have only one `MarkerTypeCatalog` statement.

Syntax

`<MarkerTypeCatalog`	
`<MTypeName` *`string`*`>`	Marker name, as it appears in the Marker Type popup menu of the Marker dialog box.
`>#end of MarkerTypeCatalog`	End of `MarkerTypeCatalog` statement

Marker statement

The `Marker` statement inserts a marker. It must appear in a `ParaLine` statement.

For version 5.5 of MIF and later, markers are identified by their names. If you open an earlier version MIF file that uses markers of type 11 through type 25, the document will show those marker numbers as the marker names. For MIF version 5.5 or later, `MType` numbers are still assigned for backward compatibility, but the assignment of numbers is fairly arbitrary. If the document includes more than 15 custom markers (Type 11 through Type 25), then the extra custom markers will be assigned `<MType 25>`.

Syntax

`<Marker`	
`<Unique` *`ID`*`>`	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
`<MType` *`integer`*`>`	Marker type number (for list of allowed values, see "Usage," next). Marker type numbers are not used for the current versions of FrameMaker, but they are included for backward compatibility
`<MTypeName` *`string`*`>`	Marker name, as it appears in the Marker Type popup menu of the Marker dialog box
`<MText` *`string`*`>`	Marker text string
`<MCurrPage` *`integer`*`>`	Current page of marker assigned when FrameMaker generates a file; ignored when FrameMaker reads or imports a MIF file
`>`	End of `Marker` statement

Usage

Marker type numbers correspond to the marker names in the Marker window as follows.

This number	Represents this marker name
0	Header/Footer $1
1	Header/Footer $2
2	Index
3	Comment
4	Subject
5	Author
6	Glossary

This number	Represents this marker name
7	Equation
8	Hypertext
9	X-Ref
10	Conditional Text
11 through 25	Type 11 through Type 25, for versions of FrameMaker earlier than 5.5. If more than 25 markers are defined for the document, all extra markers are assigned the number 25.

In UNIX versions, you can change the default marker names. For more information, see the online manual, *Customizing FrameMaker.*

XRef statement

The `XRef` statement marks a cross-reference in text. It must appear in a `ParaLine` statement.

Syntax

`<XRef`	
`<Unique` *`ID`*`>`	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the FDK client and should not be used by filters
`<XRefName` *`tagstring`*`>`	Name of cross-reference format (see "XRefFormats and XRefFormat statements" on page 88)
`<XRefLastUpdate` *`seconds microseconds`*`>`	Specifies the time when the cross-reference was last updated; time is measured in the number of seconds and microseconds that have passed since January 1, 1970
`<XRefLocked` *`boolean`*`>`	`Yes` means the cross-reference is part of a text inset that obtains its formatting information from the source document
`<XRefSrcText` *`string`*`>`	Text to search for
`<XRefSrcIsElem` *`boolean`*`>`	`Yes` means the source of the cross-reference is an element from a structured document
`<XRefSrcFile` *`pathname`*`>`	Device-independent pathname of file in which to search for source text (for *`pathname`* syntax, see page 7)
`<XRefSrcElemNonUniqueId` *`string`*`>`	A string specifying the 'id' attribute of the source element, in case it is not a unique ID
`<XRefAltText` *`string`*`>`	Alternate display text
`<XRefApiClient`	The client for the cross-reference. Contains `XRefClientName` and `XRefClientType`
`<XRefClientName` *`string`*`>`	The registered name of the client that created the cross-reference
`<XRefClientType` *`string`*`>`	The type of the client that created the cross-reference
`>`	End of `XRefApiClient` statement
`>`	End of `XRef` statement
`<Font…>`	Embedded character change for the following cross-reference text (see page 67)

<String *string*>	Text of cross-reference
<XRefEnd>	End of cross-reference

Usage

The XRef statement marks where a cross-reference appears in text. The XRefName statement applies a format to the cross-reference text; its string argument must match the name of the format provided by an XRefFormat statement.

The XRefSrcText statement identifies the cross-reference source. If the source text is in a separate file, the XRefSrcFile statement provides a device-independent filename. You can omit it or give it an empty string argument if the cross-reference source is in the same file.

The XRefEnd statement marks the end of the cross-reference.

Any String or Char statements between the XRef and XRefEnd statements represent the actual text of the cross-reference. These intermediary statements are optional.

For an example of a cross-reference in MIF, see "Creating cross-references" on page 37.

The XRefLocked statement is used for text insets that retain formatting information from the source document.

If the <XRefLocked Yes> statement appears in a specific cross-reference, that cross-reference is part of a text inset that retains formatting information from the source document. The cross-reference is not affected by global formatting performed on the document.

If the <XRefLocked No> statement appears in a specific cross-reference, that cross-reference is not part of a text inset, or is part of a text inset that reads formatting information from the current document. The cross-reference is affected by global formatting performed on the document.

For more information about text insets, see "Text insets (text imported by reference)," next.

Text insets (text imported by reference)

In a FrameMaker document, text can be imported by reference from another file. When the text in the original file is modified, the imported text in the FrameMaker document is updated with changes. Text imported by reference is called a *text inset*. In a MIF file, text insets are defined by the TextInset statement.

A TextInset statement appears in the ParaLine statement representing the location of the text being imported. When text is imported by reference, the resulting text inset can be formatted either as regular text or as a table.

The source file (from which the text is imported) can be a FrameMaker document or any other kind of text file. The source file can also be a file that is created, maintained, and updated by an FDK client (a program created with the Frame Developer's Kit.

TextInset statement

The TextInset statement defines text that has been imported by reference. A TextInset statement appears in a ParaLine statement.

Syntax

<TextInset	
<Unique *num*>	Unique ID number assigned by FrameMaker
<TiName *string*>	Specifies a name for the text inset that may be assigned by an FDK client or by this statement in a MIF file; FrameMaker does not automatically assign a name for the text inset
<TiSrcFile *pathname*>	Specifies the source file with a device-independent filename (for *pathname* syntax, see page 7)
<TiAutoUpdate *boolean*>	Yes specifies that the text inset is updated automatically when the source file changes
<TiLastUpdate *seconds microseconds*>	Specifies the time when the text inset was last updated; time is measured in the number of seconds and microseconds that have passed since January 1, 1970
<TiImportHint *string*>	Identifies the filter used to convert the file (see "Record of the filter used to import text" on page 140)
<TiApiClient …>	Identifies the text inset as one created and maintained by an FDK client (see "TiApiClient statement" on page 141)
<TiFlow …>	Identifies the text inset as an imported text flow from another document (see "TiFlow statement" on page 142)
<TiText …>	Identifies the text inset as an imported text file (see "TiText statement" on page 143)
<TiTextTable …>	Identifies the text inset as text imported into a table (see "TiTextTable statement" on page 143)
>	End of TextInset statement
…(*Free-form text*)	Para statements containing and describing the imported text (see "Para statement" on page 132)
<TextInsetEnd>	End of imported text

Usage

All text insets require information about the source file and the imported text. The information is used to update the text inset when changes are made to the original file.

There are several different types of text insets. The type of the text inset is identified and described by a substatement:

- Text created and maintained by an FDK client is described by the TiApiClient substatement. For information on the statement, see the section "TiApiClient statement" on page 141.
- A text flow imported from another FrameMaker document or from a document filtered by FrameMaker is described by the TiFlow substatement. For information on the statement, see the section "TiFlow statement" on page 142.
- Plain text imported by reference is described by the TiText substatement. For information on the statement, see the section "TiText statement" on page 143.
- Text imported into a tabular format is described by the TiTextTable substatement. For information on the statement, see the section "TiTextTable statement" on page 143.

Usage of some of the aspects of the TextInset statement is described in the following section.

Record of the filter used to import text

The `TextInset` statement contains a record to identify the filter that was used to import text by reference. FrameMaker uses the record to find the correct filter to use when updating the text inset.

The record is specified in the `TiImportHint` statement and uses the following syntax:

record_vers vendor format_id platform filter_vers filter_name

Note that the fields in the record are not separated by spaces. For example:

```
`0001XTNDWDBNMACP0002MS Word 4,5'
```

In this example, `0001` is the record version; `XTND` is the vendor; `WDBN` is the format id; `MACP` is the platform; `0002` is the filter version; and `MS Word 4,5` is the filter name. The rest of this section describes each field in the record.

record_vers is the version on the record (for example, `0001`).

vendor is a code specifying the filter's vendor. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
'PGRF'	Built-in FrameMaker filters
'FAPI'	External FDK client filter
'FFLT'	External FrameMaker filters
'IMAG'	External ImageMark filters
'XTND'	External XTND filters

Note that this is not a comprehensive list of codes. Codes may be added to this list by Adobe or by developers at your site.

format_id is a code specifying the format that the filter translates. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
'WDBN'	Microsoft Word compound document
'WPBN'	WordPerfect compound document
'RTF'	Microsoft's RTF compound document
'IAF'	Interleaf compound document
'MIF'	Maker Interchange Format
'MRTF'	MIF to RTF export
'MIAF'	MIF to IAF export
'MWPB'	MIF to WordPerfect export
'TRFF'	`troff` to MIF (UNIX only)
'MML'	Maker Mark-up Language
'CVBN'	Corel Ventura compound document (Windows)
'DCA'	DCA to MIF (UNIX)
'TEXT'	Plain text
'TXIS'	Text ISO Latin 1

Code	Description
'TXRM'	Text Roman 8
'TANS'	Text ANSI
'TASC'	Text ASCII
'TSJS'	Shift-JIS
'TBG5'	Big5
'TGB'	GB-2312
'TKOR'	Korean
'TUT8'	UTF-8
'TU1B'	UTF-16BE
'TU1L'	UTF-16LE
'TU3B'	UTF-32BE
'TU3L'	UTF-32LE

Note that this is not a comprehensive list of codes. Codes may be added to this list by Adobe or by developers at your site.

platform is a code specifying the platform on which the filter was run. The code is a string of four characters. The following table lists some of the possible codes.

Code	Description
'WINT'	Windows NT
'WIN3'	Windows 3.1
'WIN4'	Windows 95
'UNIX'	Generic X/11 (Sun, HP)

filter_vers is a string of four characters identifying the version of the filter on that platform. For example, version 1.0 of a filter is represented by the string `1.0 '.

filter_name is a text string (less than 31 characters long) that describes the filter.

TiApiClient statement

The TiApiClient statement defines a text inset created and maintained by an FDK client application.

Syntax

<TiApiClient	
<TiClientName *string*>	Specifies the name used to register the FDK client application with FrameMaker
<TiClientSource *string*>	Specifies the location of the source file for the text inset
<TiClientType *string*>	Specifies the type of the source file
<TiClientData *string*>	Specifies additional data that can be used by an FDK client (for example, SQL query information)

>	End of TiApiClient statement

Usage

When updating text insets, the FDK client can use the TiClientName substatement to determine if it should update a given text inset.

If the FDK client requires additional information, the client can store the information in the TiClientData substatement. For example, if the FDK client queries a database for text, the SQL query can be stored in the TiClientData substatement.

TiFlow statement

The TiFlow statement defines a text flow that is imported by reference from a FrameMaker document or a MIF file. The statement also defines imported text from other formatted documents that FrameMaker can filter (for example, a Microsoft Word document).

Syntax

<TiFlow	
<TiFormatting *keyword*>	Specifies which document formats are used for the text inset *keyword* can be one of: TiSource TiEnclosing TiPlainText
<TiMainFlow *boolean*>	Yes specifies that the text inset is imported from the main flow of the source document; No specifies that the text inset is imported from a different flow
<TiPageSpace *keyword*>	If the text inset is not imported from the main flow, specifies whether the text inset is imported from a flow in the body page or the reference page of the source document *keyword* can be one of: BodyPage ReferencePage
<TiFlowName *string*>	If the text inset is not imported from the main flow, specifies the tag of the flow to import; if the source file is an edition, set to `Windows edition'
<TiFormatRemoveOverrides *boolean*>	When reformatting to use the current document's formats, Yes specifies that format overrides are removed
<TiFormatRemovePageBreaks *boolean*>	When reformatting to use the current document's formats, Yes specifies that manual page breaks are removed
>	End of TiFlow statement

Usage

If the imported text flow is not the main flow of the source document, the TiPageSpace and TiFlowName substatements identify the flow in the source document that serves as the imported text flow.

Text imported from another document can obtain formatting information from the original document (if the TiFormatting statement is set to TiSource) or from the current document (if the TiFormatting statement is set to TiEnclosing):

- If the imported text flow is reformatted to use the current document's formats, the `TiFormatRemoveOverrides` substatement specifies whether or not format overrides in the text are removed, and the `TiFormatRemovePage-Breaks` substatement specifies whether or not manual page breaks in the text are removed.
- If the imported text flow retains the formatting of the source document, the paragraph, character, table, variable, and cross-reference formats used in the inset are marked with special MIF statements to indicate that these formats should not be affected by global updates. These statements are `PgfLocked`, `FLocked`, `TblLocked`, `VariableLocked`, and `XRefLocked`, respectively. The MIF statements appear under the descriptions of these formats.

Plain text formatting can also be used, if the `TiFormatting` statement is set to `TiPlainText`.

TiText statement

The `TiText` statement defines a text file imported by reference. It appears in a `TextInset` statement.

Syntax

<TiText	
<TiEOLisEOP *boolean*>	`Yes` specifies that the end of the line marks the end of a paragraph; `No` specifies that a blank line identifies the end of a paragraph

<TiTxtEncoding *keyword*>	Specifies the text encoding for the source file
	
	keyword can be one of:
	`TiIsoLatin`
	`TiASCII`
	`TiANSI`
	`TiJIS`
	`TiShiftJIS`
	`TiEUC`
	`TiBig5`
	`TIEUCCNS`
	`TiGB`
	`TiHZ`
	`TiKorean`
	`TiUTF8`
	`TiUTF16BE`
	`TiUTF16LE`
	`TiUTF32BE`
	`TiUTF32LE`

>	End of `TiText` statement

TiTextTable statement

The `TiTextTable` statement defines imported text formatted as a table. It appears in a `TextInset` statement.

Syntax

<TiTextTable	
<TiTblTag *string*>	Specifies the name of the table format used for the table
<TiTblIsByRow *boolean*>	`Yes` specifies that each paragraph in the imported text is converted to a row of table cells; `No` specifies that each paragraph in the imported text is converted to a table cell

<TiTblNumCols *num*>	If each paragraph is converted to a separate cell, specifies the number of columns in the table
<TiTblSep *string*>	If each paragraph is converted to a row of cells, specifies the character used to indicate the contents of each cell
<TiTblNumSep *num*>	If characters are used to indicate the contents of each cell, specifies the number of these characters used as a single divider
<TiTblNumHdrRows *num*>	Specifies the number of heading rows in the table
<TiTblHeadersEmpty *boolean*>	`Yes` indicates that the imported text is not inserted in the heading rows
<TiTblTxtEncoding *keyword*>	Specifies the text encoding for the source file *keyword* can be one of: TiIsoLatin TiASCII TiANSI TiJIS TiShiftJIS TiEUC TiBig5 TIEUCCNS TiGB TiHZ TiKorean TiUTF8 TiUTF16BE TiUTF16LE TiUTF32BE TiUTF32LE
>	End of `TiTextTable` statement

Usage

When imported text is converted to a tabular format, each paragraph can be converted into either a cell or a row of cells:

- If each paragraph is converted to a table cell, the `TiTblIsByRow` substatement is set to `No`. The number of columns in the table is specified by the `TiTblNumCols` substatement.
- If each paragraph is converted to a row of cells, the `TiTblIsByRow` substatement is set to `Yes`. The character used in the imported text to delimit the contents of each cell is specified by the `TiTblSep` substatement, and the number of these characters used as a single divider is specified by the `TiTblNumSep` substatement.
- For example, if the imported text uses a single tab character to distinguish the contents of one table cell from the next, the following substatements are used:

```
<TiTblSep `\t'>
<TiTblNumSep 1>
```

- As another example, if the imported text uses two spaces to distinguish the contents of one table cell from the next, the following substatements are used:

```
<TiTblSep ` '>
<TiTblNumSep 2>
```

If the `TiTblNumHdrRows` substatement is not set to `0`, the table has header rows. If the `TiTblHeadersEmpty` substatement is set to `No`, these rows are filled with imported text.

Chapter 4: MIF Book File Statements

MIF book file overview

The following table lists the main statements in a MIF book file in the order that Adobe® FrameMaker® writes them. You should follow the same order that FrameMaker uses, with the exception of the macro statements and control statements, which can appear anywhere at the top level of a file. Each statement, except the `Book` statement, is optional. Most main statements use substatements to describe objects and their properties.

Section	Description
`Book`	Labels the file as a MIF book file. The `Book` statement is required and must be the first statement in the file.
`Macro statements`	Defines macros with a `define` statement and reads in files with an `include` statement. These statements can appear anywhere at the top level.
`Control statements`	Establishes the default units in a `Units` statement, the debugging setting in a `Verbose` statement, and comments in a `Comment` statement. These statements can appear anywhere at the top level.
`BWindowRect`	Specifies position of book window on the screen.
`View only statements`	Specify whether the book is View Only, and how to display View Only book windows
`BDisplayText`	Specifies the type of text to display in the book window for each book component icon.
`PDF statements`	Specify document info entries and how to handle named destinations when you save the book as PDF
`BookComponent`	Provides the setup information for each file in the book.
`Color Catalog`	The color definitions of each document in the book.
`Condition Catalog`	Defines the condition tags of each document in the book.
`Combined Font Catalog`	Defines the combined fonts of each document in the book.
`FontCatalog`	Defines the character formats of each document in the book. The `FontCatalog` statement contains a series of `Font` statements that define the tags that appear in the Character Catalog of generated files.
`PgfCatalog`	Defines the paragraph formats of each document in the book. The `PgfCatalog` statement contains a series of `Pgf` statements that define the tags that appear in the Include and Don't Include scroll lists of the setup dialog boxes for generated files.
`BookXRef`	Names and defines the book's internal cross-references. The `BookXRef` statement contains cross-reference definitions in `XRefDef` statements, cross-reference text in `XRefSrcText` statements, and the source filename in `XRefSrcFile` statements.
`BookUpdateReferences`	Specifies whether or not cross-references and text insets are automatically updated when the book file is opened.
`WEBDAV statements`	Specifies whether or not a book is marked as managed content on the WebDAV server.

MIF book file identification line

The MIF book file identification line must be the first line of the file with no leading white space.

Syntax

`<Book version> # comment`

The *version* argument indicates the version number of the MIF language used in the file, and *comment* is a comment showing the name and version number of the program that generated the file.

For example, a MIF book file saved in version 9 of FrameMaker begins with the following line:

```
<Book 2015> # Generated by version 9.0 of FrameMaker
```

MIF is compatible across versions, so a MIF interpreter should be able to parse any MIF file, although the results can sometimes differ from the user's intentions.

A MIF book file identification line is the only statement required in a MIF book file.

Book statements

A MIF file for a book contains statements specific to books (`BWindowRect`, `BookComponent`, `BookXRef`, and `BookUpdateReferences`), plus the following statements, which can also occur in a MIF file for a document: `Comment`, `Units`, `Verbose`, `PgfCatalog`, and `FontCatalog`, `ColorCatalog`, and `ConditionCatalog`.

BWindowRect statement

The `BWindowRect` statement defines the position of the book window on the screen. It can appear anywhere in the file but normally appears just after the `Book` statement.

Syntax

`<BWindowRect X Y W H>`	Book window placement on screen

PDF statements

The `PDFBookInfo` statement specifies the information to include in the Document Info dictionary when you save the book as PDF. Each data entry consists of one `Key` statement, followed by at least one `Value` statement; you can include as many `Value` statements as you like. FrameMaker ignores any `Key` that does not have at least one `Value` following it. MIF does not represent entries for `Creator`, `Creation Date`, or `Modification Date`.

For additional information and an example of the syntax for the `Key` and `Value` statements, see "PDF Document Info" on page 89

Syntax

`<PDFBookInfo`	Specifies the information that appears in the File Info dictionary when you save the book as PDF Each Document Info entry consists of one `Key` statement followed by at least one `Value` statement.

<Key string>	A string of up to 255 ASCII characters that represents the name of a Document Info field; in PDF the name of a File Info field must be 126 characters or less. Represent non-printable characters via #HH, where # identifies a hexadecimal representation of a character, and HH is the hexadecimal value for the character. For example, use #23 to represent the "#" character. Zero-value hex -codes (#00) are illegal. For more information, see"PDF Document Info" on page 89.
<Value string>	A string of up to 255 ASCII characters that represents the value of a Document Info field; because a single MIF string contains no more than 255 ASCII characters, you can use more than one Value statement for a given Key A Value can include Unicode characters; represent Unicode characters via &#xHHHH;, where &#x opens the character code, the ";" character closes the character code, and HHHH are as many hexadecimal values as are required to represent the character. For more information, see "PDF Document Info" on page 89.
...	You can repeat paired groupings of Key and Value statements
>	End of PDFBookInfo statement

The BookFileInfostatement stores encoded packets of information (XMP data) that corresponds with values of fields in the File Info dialog box. This statement can only appear in the Book statement.

Syntax°

<BookFileInfo>	Specifies the same information that appears in <PDFBookInfo>, except it expresses these values as encoded data. You should not try to edit this data. BookFileInfo also represents the values of the default fields for Creator, Creation Date, and MetaData Date. For more information, see "Document File Info" on page 89.
<*encoded*>	XMP information as encoded data which is generated by FrameMaker. This information corresponds to the values set in the File Info dialog box. For any book, there can be an arbitrary number of XMP statements.
...	
>	End of BookFileInfo

XML book statements

In versions 7.0 and later, FrameMaker supports XML import and export. The following statements store information necessary to properly save a book as XML.

Syntax

<BXmlVersion string>	The XML version that was specified in the XML declaration when the XML file was opened
<BXmlEncoding string>	The XML encoding parameter that was specified in the XML declaration when the XML file was opened
<BXmlStandAlone int>	The XML standalone parameter that was specified in the XML declaration when the XML file was opened—determines whether or not the XML document requires a DTD

<BXmlStyleSheet string>	The path or URI to the stylesheet that was specified for the XML file, plus the type parameter specifying the type of stylesheet

View only book statements

In versions 6.0 and later, a book can be View Only. The following statements indicate whether the book is View Only, and how to display the book window when it is View Only.

Syntax

<BViewOnly boolean>	Yes specifies View Only book (locked)
<BViewOnlyWinBorders boolean>	No suppresses display of scroll bars and border buttons in book window of View Only book
<BViewOnlyWinMenubar boolean>	No suppresses display of book window menu bar in View Only book (Unix only)
<BViewOnlyPopup boolean>	No suppresses display of book context menus in View Only book
<BViewOnlyNoOp 0xnnn>	Disables a command in a View Only document; command is specified by hex function code (see page 48)

BDisplayText statement

The BDisplayText statement defines the type of text to display in the book window next to the book component icons. It can appear anywhere in the file but normally appears just after the book's View Only statements.

Syntax

<BWindowRect *X Y W H*>	Book window placement on screen

<BDisplayText keyword>	The type of text to display next to component icons in the book window; keyword can be one of: AsFilename; displays the filename of the book component in the book window. AsText; displays a text snippet from the first paragraph of the component in the book window

BookComponent statement

The BookComponent statement contains the setup information for a folder, group, document, or generated file in a book. The BookComponent statements must precede all other statements that represent book content. The order of BookComponent statements determines the order of the documents in the book.

If the BookComponentType is either a folder or a group, the following BookComponent statements should begin under a BeginFolder and EndFolder statements or BeginGroup and EndGroup statements. The sequence should be as follows.

```
<BookComponent
    <BookComponentType FolderBookComponent>
    <ComponentTitle 'Folder Name'>
    <Expanded Yes>
    <ExcludeComponent No>
    ...
    <ComponentTemplateFilePath 'folder_template.fm'>
> # end of BookComponent
<BeginFolder> or <BeginGroup>
    <BookComponent
        <BookComponentType GeneralBookComponent>
        ...
    > # end of BookComponent for file 1
        #There can be multiple BookComponent statements within a BeginFolder and EndFolder
         statements.
<EndFolder> or <EndGroup>
```

You specify the setup information as substatements nested within the overall book component statement. A `BookComponent` statement doesn't need all these substatements, which can occur in any order. A `BookComponent` statement can contain one or more `DeriveTag` statements.

Folder components	
<BeginFolder>	If the `BookComponentType` is `FolderBookComponent` then this tag appears before the following `BookComponent` tag.
<ComponentTitle>	The name for the folder or group.
<EndFolder>	`EndFolder` indicates the end of the folder started with the immediately previous `BeginFolder` statement.
Group components	
<BeginGroup>	If the `BookComponentType` is `GroupBookComponent` then this tag appears before the following `BookComponent` tag.
<EndGroup>	`EndGroup` indicates the end of the group started with the immediately previous `BeginGroup` statement.

Syntax

`<BookComponent`	Book components
`<FileName pathname>`	A document or generated file in the book (for pathname syntax, see page 7)
`<DisplayText string>`	The text to display in the book window next to the icon for this component; FrameMaker displays this text when `BDisplayText` is set to `AsText` (see "<BDisplayText keyword>" on page 148).
`<BookComponentType string>`	The type of book component `GroupBookComponent` (group) `FolderBookComponent` (folder) `GeneralBookComponent` (regular component)
`<Expanded boolean>`	`Yes` expands the node in case of a hierarchy
`<ExcludeComponent boolean>`	`Yes` exludes the component
Generated components	
`<FileNameSuffix string>`	Filename suffix added to generated file

<DeriveType keyword>	Type of generated file *keyword* can be one of: AML (alphabetic marker list) APL (alphabetic paragraph list) IDX (index) IOA (author index) IOM (index of markers) IOS (subject index) IR (index of references) LOF (list of figures) LOM (list of markers) LOP (list of paragraphs) LOT (list of tables) LR (list of references) TOC (table of contents) MLT (master list template)
<DeriveTag tagstring>	Tags to include in generated file
<DeriveLinks boolean>	Yes automatically creates hypertext links in generated files
Book component pagination and numbering properties	
<StartPageSide keyword>	The page side on which to start *keyword* can be one of: ReadFromFile (default) NextAvailableSide StartLeftSide StartRightSide
Volume numbering	
<VolumeNumStart integer>	Starting volume number
<VolumeNumStyle keyword>	Style of volume numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<VolumeNumText string>	When VolumeNumStyle is set to Custom, this is the string to use

<VolNumComputeMethod keyword>	Volume numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Chapter numbering	
<ChapterNumStart integer>	Starting chapter number
<ChapterNumStyle keyword>	Style of chapter numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<ChapterNumText string>	When ChapterNumStyle is set to Custom, this is the string to use
<ChapterNumComputeMethod keyword>	Chapter numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Section numbering	
<SectionNumStart integer>	Starting section number

<SectionNumStyle keyword>	Style of section numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<SectionNumText string>	When SectionNumStyle is set to Custom, this is the string to use
<SectionNumComputeMethod keyword>	Section numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Sub section numbering	
<SubSectionNumStart integer>	Starting sub section number
<SubSectionNumStyle keyword>	Style of sub section numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<SubSectionNumText string>	When SubSectionNumStyle is set to Custom, this is the string to use

<SubSectionNumComputeMethod keyword>	Sub section numbering *keyword* can be one of: StartNumbering (restart numbering) ContinueNumbering (continue numbering from previous component) UseSameNumbering (use the same numbering as previous component) ReadFromFile (use numbering set for the component's document)
Page numbering	
<ContPageNum boolean>	Yes continues page numbering from the previous file in the book
<PageNumStart integer>	Starting page number
<PageNumStyle keyword>	Style of page numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu
<PageNumbering keyword>	Page numbering *keyword* can be one of: Continue (default) Restart ReadFromFile
Paragraph numbering	
<PgfNumbering keyword>	Paragraph numbering *keyword* can be one of: Continue (default) Restart ReadFromFile
Footnote numbering	
<BFNoteStartNum integer>	Starting number for footnote numbering

<BFNoteNumStyle keyword>	Style of footnote numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<BFNoteLabels string>	When BFNoteNumStyle is set to Custom, this is the string to use
<BFNoteComputeMethod keyword>	Footnote numbering *keyword* can be one of: Continue (continue numbering from previous component in book) Restart (restart numbering; typically to restart per flow, according to BFNoteRestart setting) PerPage (restart footnote numbering for each page; overrides BFNoteRestart setting) ReadFromFile (use numbering set for the component's document)
<BFNoteRestart keyword>	When to restart numbering, if BFNoteComputeMethod is set to Restart *keyword* can be one of: PerFlow (restart footnote numbering for each flow in the document PerPage (restart footnote numbering for each page)
Table footnote numbering	
<BTblFNoteNumStyle keyword>	Style of table footnote numbering *keyword* can be one of: IndicNumeric FarsiNumeric HebrewNumeric AbjadNumeric AlifbataNumeric UCRoman LCRoman UCAlpha LCAlpha KanjiNumeric ZenArabic ZenUCAlpha ZenLCAlpha Kanjikazu BusinessKazu Custom
<BTblFNoteLabels string>	When BTblFNoteNumStyle is set to Custom, this is the string to use

<BTblFNoteComputeMethod keyword>	Table footnote numbering; either value causes the component to read the numbering style from its document *keyword* can be one of: Restart (use numbering style specified in the component) ReadFromFile (use numbering style set for the component's document)
Book component defaults	
<DefaultPrint boolean>	Yes adds file to Print scroll list in Print Files in Book dialog box (file is printed); saved for compatibility with versions earlier than 6.0
<DefaultApply boolean>	Yes adds file to Update scroll list in the Import Formats dialog box (file is updated); saved for compatibility with versions earlier than 6.0
<DefaultDerive boolean>	Yes adds file to Generate scroll list in the Generate/Update Book dialog box
<NumPages integer>	The number of pages in the components document, as calculated the last time the book was updated
<ComponentIsDitaMap boolean>	Yes if the component file path is a DITA map
Book autonumbering	
<BookInitialAutoNums	Provides a starting value for the autonumber series in a book.
<FlowTag *string*>	Specifies flow that the book uses to number the series
<Series *string*>	Specifies autonumber series
<NumCounter *integer*>	Initializes autonumber counter
<NumCounter ...>	Additional statements as needed
...	
>	End of AutoNumSeries statement
Folder or group properties	
<ComponentApplication string>	Name of the application for a folder, template, or XML file
<ComponentTemplateFilePath string>	The path and filename of the folder template
Book conditional tags	
<AllConditionTags	Container object that contains objects of type TagName
<TagName *string*>	Name of the managed condition tag
>	End of AllConditionTags statement
<ShownConditionalTags	Container object that contains objects of type TagName
<TagName *string*>	Name of the managed condition tag
>	End of ShownConditionalTags statement
>	End of BookComponent statement

BookXRef statement

The `BookXRef` statement defines the cross-reference formats for the book.

Syntax

`<BookXRef`	
`<XRefDef` *`string`*`>`	Cross-reference format definition
`<XRefSrcText` *`string`*`>`	Text for which to search
`<XRefSrcIsElem` *`boolean`*`>`	`Yes` means the source of the cross-reference is an element from a structured document
`<XRefSrcFile` *`pathname`*`>`	File in which to search for source text (for *`pathname`* syntax, see page 7)
`<XRefSrcElemNonUniqueId` *`string`*`>`	A string specifying the 'id' attribute of the source element, in case it is not a unique ID
`<XRefAltText` *`string`*`>`	Alternate display text
`>`	End of `BookXRef` statement

BookUpdateReferences statement

The `BookUpdateReferences` statement specifies whether or not cross-references and text insets are automatically updated when the book file is opened.

Syntax

`<BookUpdateReferences` *`boolean`*`>`	`Yes` specifies that cross-references and text insets are automatically updated when the book file is opened

WEBDAV statements

The BookServerURL and BookServerState MIF statements mark a book as managed content from the WebDAV perspective.

Syntax

`<BookServerURL` *`string`*`>`	URL of the MIF book file on the WEBDAV Server. All http path values are valid. Example: `<BookServerUrl `http://mikej-xp/joewebdav/myfile.book.mif'> # http://mikej-xp/joewebdav is the path of the server.`
`<BookServerState` *`keyword checked-out checkedin`*`>`	Indicates whether a book is checked in or checked out on the WebDAV server. Example: `<BookServerState CheckedIn>`

Chapter 5: MIF Statements for Structured Documents and Books

This chapter describes the MIF statements that define structured documents created with Adobe® FrameMaker®. For more information about creating and editing structured documents, see the *FrameMaker User Guide*.

Structural element definitions

A structured document is divided into logical units called *structural elements*. Elements have tags (or names) that indicate their role in the document. For example, a document might contain Section, Para, List, and Item elements. Each element has a definition that specifies its valid contents (such as text and graphics). A structured template specifies a document's elements, and the correct order of elements and text in the document.

There are two basic groups of structure elements:

- Containers, tables and footnotes, which can hold text and other elements.
- Object elements, such as graphic frames, equations, markers, system variables, and cross-references. An object element holds one of its specified type of object and nothing more.

Tables belong to both groups of elements. Although they can contain other elements (table parts such as rows and cells), tables are also object elements.

In a MIF file, an element definition is defined by an `ElementDef` statement. Element definitions are stored in the Element Catalog, which is defined by the `ElementDefCatalog` statement. Within a text flow, elements are indicated by `ElementBegin` and `ElementEnd` statements.

When FrameMaker reads a MIF file that does not support structure, they strip MIF statements for structure, such as `ElementBegin`, `ElementEnd`, and `ElementDefCatalog` statements.

ElementDefCatalog statement

The `ElementDefCatalog` statement defines the contents of the Element Catalog. A document or book file can have only one `ElementDefCatalog` statement which must appear at the top level in the order given in "MIF file layout" on page 10.

Syntax

`<ElementDefCatalog`	Begin Element Catalog
`  <ElementDef...>`	Defines an element (see "ElementDef statement," next)
`  <ElementDef...>`	Additional statements as needed
`  ...`	
`>`	End of `ElementDefCatalog` statement

ElementDef statement

The `ElementDef` statement creates an element definition, which specifies an element's tag name, content rules, and optional format rules. It must appear within an `ElementDefCatalog` statement.

Syntax

`<ElementDef`	Begin element definition
`<EDTag` *tagstring*`>`	Element tag name

`<EDObject` *keyword*`>`	Type of formatter object represented by the element
	
	keyword can be one of:
	`EDContainer`
	`EDEquation`
	`EDFootnote`
	`EDGraphic`
	`EDMarker`
	`EDTable`
	`EDTblTitle`
	`EDTblHeading`
	`EDTblBody`
	`EDTblFooting`
	`EDTblRow`
	`EDTblCell`
	`EDSystemVariable`
	`EDXRef`
	
	`EDContainer` identifies a container element; all other values identify object (non-container) elements

`<EDValidHighestLevel` *boolean*`>`	`Yes` indicates element can be used as the highest level element for a flow; only a container element is allowed to be the highest level element
`<EDGeneralRule` *string*`>`	The general rule for the element; the following types of elements can have general rules: containers, tables, table parts (table titles, headings, bodies, footings, rows, and cells), and footnotes
`<EDExclusions`	List of excluded elements
`<Exclusion` *tagstring*`>`	Tag of excluded element
`<Exclusion` *tagstring*`>`	Additional statements as needed
...	
`>`	End of `EDExclusions` statement
`<EDInclusions`	List of included elements
`<Inclusion` *tagstring*`>`	Tag of included element
`<Inclusion` *tagstring*`>`	Additional statements as needed
...	
`>`	End of `EDInclusions` statement
`<EDAlsoInsert`	List of elements that are automatically inserted in a container element when the element is initially added
`<AlsoInsert` *tagstring*`>`	Tag of inserted element

<AlsoInsert *tagstring*>	Additional statements as needed
...	
>	End of EDAlsoInsert statement
<EDInitialTablePattern *string*>	List of the tags of table child elements that are automatically created when a table is inserted Valid only if EDObject is one of the following: EDTable EDTblHeading EDTblBody EDTblFooting EDTblRow EDTblCell
<EDAttrDefinitions	List of attribute definitions
<EDAttrDef...>	Definition of attribute (see "Attribute definitions" on page 160)
<EDAttrDef...>	Additional statements as needed
...	
>	End of EDAttrDefinitions statement
<EDPgfFormat *string*>	Paragraph format of the element
<EDStyleFormat *string*>	Style format of the element
<EDTextFormatRules...>	See "EDTextFormatRules statement" on page 162
<EDObjectFormatRules...>	See "EDObjectFormatRules statement" on page 162
<EDPrefixRules...>	See "EDPrefixRules statement" on page 163
<EDSuffixRules...>	See "EDSuffixRules statement" on page 163
<EDStartElementRules...>	See "EDStartElementRules statement" on page 164
<EDEndElementRules...>	See "EDEndElementRules statement" on page 164
<EDBannerText string>	The banner text that appears inside a new element instance
<EDDescriptiveTag *string*>	Description of the element tag that appears next to the element in the element catalog
<EDComments *string*>	Comments for the element definition
>	End of ElementDef statement

Usage

The element name can contain any characters from the FrameMaker character set except the following:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

Content rules

The content rule for a container element consists of the following statements:

- A required <EDObject EDContainer> statement specifies the element type.
- A required EDGeneralRule statement specifies what the element can contain and in what order the element's contents can appear.

- An optional `EDExclusions` statement specifies elements that cannot appear in the defined element or in its descendants.
- An optional `EDInclusions` statement specifies elements that can appear anywhere in the defined element or in its descendants.

The general rule specification must follow the conventions for data in a MIF string. If a general rule contains angle brackets (<>), the right angle bracket must be preceded by a backslash in the MIF string. For example, an element that can contain text might have the following general rule:

```
<EDGeneralRule `<TEXT\>'>
```

If you don't provide a general rule statement for a container element, the MIF interpreter applies the default rule `<ANY>`. The rule means that any element or text is allowed.

The following general rule describes an element that must contain at least one element named Item.

```
<ElementDef
    <EDTag `BulletList'>
    <EDValidHighestLevel No >
    <EDGeneralRule `Item+'>
    <EDObject EDContainer >
>           # end of ElementDef
```

For more information about content rules, see the online manual *FrameMaker Structure Application Developer's Guide*.

Attribute definitions

Element definitions can specify *attribute definitions*, which describe attributes (information stored with an element other than its content). The definition of an attribute can specify that the attribute is required for all elements with the element definition. It can also provide a list of the values the attribute can have and a default value.

EDAttrDef statement

The `EDAttrDef` statement defines the formatting properties to be applied to a container, table, table child, or footnote element in different contexts. It must appear in an `ElementDef` statement.

Syntax

`<EDAttrDef`	Begin attribute definition
`<EDAttrName` *string*`>`	Attribute name

<EDAttrType *keyword*>	Attribute type *keyword* can be one of: FAttrChoice: a value from a list of choices FAttrInt: a signed whole number (optionally restricted to a range of values) FAttrInts: one or more integers (optionally restricted to a range of values) FAttrReal: a real number (optionally restricted to a range of values) FAttrReals: one or more real numbers (optionally restricted to a range of values) FAttrString: an arbitrary text string FAttrStrings: one or more arbitrary text strings FAttrUniqueId: a string that uniquely identifies the element FAttrUniqueIdRef: a reference to a UniqueID attribute FAttrUniqueIdRefs: one or more references to a UniqueID attribute
<EDAttrRequired *boolean*>	Yes means the attribute is required
<EDAttrReadOnly *boolean*>	Yes means the attribute is read-only
<EDAttrHidden *boolean*>	Yes means the attribute is hidden and will not appear in the Structure view or in the Edit Attributes dialog box
<EDAttrChoices	The choices, if the attribute type is FAttrChoice
<EDAttrChoice *string*>	A choice
<EDAttrChoice *string*>	Additional statements as needed
...	
>	End of EDAttrChoices statement.
<EDAttrDefValues	The default if the attribute is not required. If the attribute type is FAttrInts, FAttrReals, FAttrStrings, or FAttrUniqueIdRefs, the default can have multiple strings
<EDAttrDefValue *string*>	A default value
<EDAttrDefValue *string*>	Additional statements as needed
...	
>	End of EDAttrDefValues statement
<EDAttrRange	Range of values the attribute is allowed to have
<EDRangeStart *string*>	The minimum value the attribute must have
<EDRangeEnd *string*>	The maximum value the attribute must have
>	End of EDAttrRange statement
>	End of EDAttrDef statement

Format rules

Format rules allow the template builder to specify the format of an element in specific circumstances. A format rule can be either a context rule or a level rule.

A *context rule* contains clauses that specify an element's formatting based on its parent and sibling elements. For example, one clause of a format rule could specify that a Para element has the FirstBody paragraph format if it is the first child of a Heading element. Another clause could specify that a Para element has the Body paragraph format in all other contexts.

A *level rule* contains clauses that specify an element's formatting on the basis of the level to which it is nested within specific types of ancestor elements. For example, one clause of a level rule could specify that a Para element appears in 12-point type if it has only one Section element among its ancestors. Another clause could specify that a Para element appears in 10-point type if there are two Section elements among its ancestors.

Element definitions contain format rules grouped into the following statements:

- `EDTextFormatRules`
- `EDObjectFormatRules`
- `EDPrefixRules`
- `EDSuffixRules`
- `EDStartElementRules`
- `EDEndElementRules`

EDTextFormatRules statement

The `EDTextFormatRules` statement defines the formatting properties to be applied to a container, table, table child, or footnote element in different contexts. It must appear in an `ElementDef` statement. An `EDTextFormatRules` statement can contain zero or more substatements describing level and context format rules.

Syntax

`<EDTextFormatRules`	Any combination of level and context format rules
`<LevelFormatRule...>`	A level format rule (see "LevelFormatRule statement" on page 165)
`<ContextFormatRule...>`	A context format rule (see "ContextFormatRule statement" on page 165)
`<ContextFormatRule...>`	Additional context format rule statements as needed
`<LevelFormatRule...>`	Additional level format rule statements as needed
...	
`>`	End of `EDTextFormatRules` statement

EDObjectFormatRules statement

The `EDObjectFormatRules` statement defines the formatting properties to be applied to a table, cross-reference, system variable, marker, graphic, or equation element in different contexts. It must appear in an `ElementDef` statement.

An `EDObjectFormatRules` statement can contain a single level format rule or a single context format rule.

Syntax

<EDObjectFormatRules	Begin object format rules (a single level format rule or a single context format rule)
<LevelFormatRule...>	A level format rule (see "LevelFormatRule statement" on page 165)
>	End of EDObjectFormatRules statement
or	
<EDObjectFormatRules	
<ContextFormatRule...>	A context format rule (see "ContextFormatRule statement" on page 165)
>	End of EDObjectFormatRules statement

EDPrefixRules statement

A *prefix* is a fixed text range that appears at the beginning of an element (before the element's content). The EDPrefixRules statement defines the formatting properties to be applied to a prefix in different contexts. It must appear in an ElementDef statement. It is valid only for container elements.

An EDPrefixRules statement can contain zero or more substatements describing level and context format rules.

Syntax

<EDPrefixRules	Begin prefix rules (any combination of level and context format rules)
<LevelFormatRule...>	A level format rule (see "LevelFormatRule statement" on page 165)
<ContextFormatRule...>	A context format rule (see "ContextFormatRule statement" on page 165)
<ContextFormatRule...>	Additional context format rule statements as needed
<LevelFormatRule...>	Additional level format rule statements as needed
...	
>	End of EDPrefixRules statement

EDSuffixRules statement

A *suffix* is a fixed text range that appears at the end of an element (after the element's content). The EDSuffixRules statement defines the formatting properties to be applied to a suffix in different contexts. It must appear in an ElementDef statement. It is valid only for container elements.

An EDSuffixRules statement can contain zero or more substatements describing level and context format rules.

Syntax

<EDSuffixRules	Begin suffix rules (any combination of level and context format rules)
<LevelFormatRule...>	A level format rule (see "LevelFormatRule statement" on page 165)
<ContextFormatRule...>	A context format rule (see "ContextFormatRule statement" on page 165)
<ContextFormatRule...>	Additional context format rule statements as needed
<LevelFormatRule...>	Additional level format rule statements as needed

...	
>	End of `EDSuffixRules` statement

EDStartElementRules statement

The `EDStartElementRules` statement defines a special set of format rules to be applied to the first paragraph in a parent element. The `EDStartElementRules` statement must appear in an `ElementDef` statement. It is valid only for container elements.

An `EDStartElementRules` statement can contain zero or more substatements describing level and context format rules.

Syntax

`<EDStartElementRules`	Begin start element rules (any combination of level and context format rules)
`<LevelFormatRule...>`	A level format rule (see "LevelFormatRule statement" on page 165)
`<ContextFormatRule...>`	A context format rule (see "ContextFormatRule statement" on page 165)
`<ContextFormatRule...>`	Additional context format rule statements as needed
`<LevelFormatRule...>`	Additional level format rule statements as needed
...	
>	End of `EDStartElementRules` statement

EDEndElementRules statement

The `EDEndElementRules` statement defines a special set of format rules to be applied to the last paragraph in a parent element. The `EDEndElementRules` statement must appear in an `ElementDef` statement. It is valid only for container elements.

An `EDEndElementRules` statement can contain zero or more substatements describing level and context format rules.

Syntax

`<EDEndElementRules`	Begin end element rules (any combination of level and context format rules)
`<LevelFormatRule...>`	A level format rule (see "LevelFormatRule statement" on page 165)
`<ContextFormatRule...>`	A context format rule (see "ContextFormatRule statement" on page 165)
`<ContextFormatRule...>`	Additional context format rule statements as needed
`<LevelFormatRule...>`	Additional level format rule statements as needed
...	
>	End of `EDEndElementRules` statement

ContextFormatRule statement

The `ContextFormatRule` statement contains clauses that specify an element's formatting on the basis of the element's parent and sibling elements. It contains an `If` statement and zero or more `ElseIf` statements. It can also contain an `Else` statement.

The `ContextFormatRule` statement must appear in a format rules statement, such as an `EDTextFormatRules` or `EDEndElementRules` statement.

Syntax

`<ContextFormatRule`	Begin context format rule
`<If…>`	An If clause (see "If, ElseIf, and Else statements" on page 166)
`<ElseIf…>`	An ElseIf clause (see "If, ElseIf, and Else statements" on page 166)
`<ElseIf…>`	Additional statements as needed
…	
`<Else…>`	An optional Else clause (see "If, ElseIf, and Else statements" on page 166)
`>`	End of `ContextFormatRule` statement

LevelFormatRule statement

The `LevelFormatRule` statement contains statements that specify an element's formatting on the basis of the level to which the element is nested within specific types of ancestor elements.

The `LevelFormatRule` statement contains a `CountElements` statement listing the tags of elements to count among the element's ancestors and a statement specifying the tag of the element at which to stop counting. The `LevelFormatRule` statement also contains an `If` statement, zero or more `ElseIf` statements, and an optional `Else` statement. The `If`, `ElseIf`, and `Else` statements define the formatting applied to the element at specified levels of nesting within the ancestor elements specified by the `CountElements` statement.

The `LevelFormatRule` statement must appear in a format rules statement, such as an `EDTextFormatRules` or `EDEndElementRules` statement.

Syntax

`<LevelFormatRule`	Begin level format rule
`<CountElements`	Optional list of elements to count among the element's ancestors
`<CountElement` *tagstring*`>`	Tag of element to count
`<CountElement` *tagstring*`>`	Additional statements as needed
…	
`>`	End of `CountElements` statement
`<StopCountingAt` *tagstring*`>`	Optional tag of element at which to stop counting
`<If…>`	An If clause (see "If, ElseIf, and Else statements" on page 166)
`<ElseIf…>`	An optional ElseIf clause (see "If, ElseIf, and Else statements" on page 166)

<ElseIf…>	Additional statements as needed
…	
<Else…>	An optional Else clause (see "If, ElseIf, and Else statements" on page 166)
>	End of LevelFormatRule

If, ElseIf, and Else statements

If, ElseIf, and Else statements specify clauses within ContextFormatRule and LevelFormatRule statements. In a ContextFormatRule statement, they specify a context and one or more statements that define how to change formatting when the context applies. If an If or ElseIf statement does not include a Context or Level statement, or the Context or Level statement contains an empty string, this indicates that the If or ElseIf statement applies in all contexts.

In a ContextFormatRule statement, If and ElseIf, and Else statements take the following form:

<If	Begin If clause
<Context *contextstring*>	String specifying a context, such as Section < Section. If this context applies to the element, the following formatting statements are used to format the element.
<*Formatting statement*>	A statement (such as a FormatTag or FmtChangeListTag statement) that specifies how to change the formatting when the Context statement applies (see "Formatting statements," next, for a list of formatting statements)
…	
>	End of If statement
<ElseIf	
<Context *contextstring*>	
<*Formatting statement*>	
…	
>	End of ElseIf statement
<Else	An optional Else clause
<*Formatting statement*>	
…	
>	End of Else statement

In a LevelFormatRule statement, If and ElseIf, and Else statements take the following form:

<If	Begin If clause
<Level *levelstring*>	String specifying a level of nesting, such as 1 or 5. If the element is nested to this level, the following formatting statements are used to format the element.

`<Formatting statement>`	A statement (such as a `FormatTag` or `FmtChangeListTag` statement) that specifies how to change the formatting when the `Level` statement applies (see "Formatting statements," next, for a list of formatting statements)
...	
`>`	End of `If` statement
`<ElseIf`	Begin ElseIf clause
`<Level levelstring>`	
`<Formatting statement>`	Additional formatting statements as needed
...	
`>`	End of `ElseIf` statement
`<Else`	An optional Else clause
`<Formatting statement>`	Additional formatting statements as needed
...	
`>`	End of `Else` statement

Formatting statements

`If`, `ElseIf`, and `Else` statements can use the following statements to specify an element's formatting:

`<IsTextRange boolean>`	`Yes` if the element is formatted as a text range instead of as a paragraph Only text format rules can include this statement.
`<FormatTag tagstring>`	The format tag. If `IsTextRange` specifies `Yes`, *`tagstring`* specifies a character format tag; otherwise, it specifies a paragraph tag, table tag, marker type, cross-reference format, or equation size Only text and object format rules can include this statement
`<FmtChangeListTag tagstring>`	The tag of a named format change list (a format change list in the format change list catalog). For more information on format change lists, see "Format change lists" on page 168 Object format rules can't include this statement
`<FmtChangeList ...>`	The definition of an unnamed format change list. For more information on format change lists, see "Format change lists" on page 168 Object format rules can't include this statement
`<ContextFormatRule ...>`	The definition of a nested context format rule
`<LevelFormatRule ...>`	The definition of a nested level format rule
`<ContextLabel labelstring>`	The context label for generated files. It cannot contain white-space characters or any of these special characters: `( ) & | , * + ? < > % [ ] = ! ; : { } "` When a user displays the Set Up dialog box to set up a generated file, the label appears next to elements to which the `If`, `ElseIf`, or `Else` statement applies Only text and object format rules can include this statement

`<ElementPrefix` *string*`>`	A string that appears before the element Only prefix rules can include this statement
`<ElementSuffix` *string*`>`	A string that appears after the element Only suffix rules can include this statement

Each `If`, `ElseIf`, and `Else` statement can include only one of the following formatting statements:

- `FormatTag`
- `FmtChangeList`
- `FmtChangeListTag`
- `ContextFormatRule`
- `LevelFormatRule`

Format change lists

A format change list specifies how a paragraph format changes when a format rule clause applies. A change list can specify a change to just a single paragraph property, or it can specify changes to a long list of properties.

A format change list can be *named* or *unnamed*. A named change list appears in the Format Change List Catalog. Format rule clauses that use a named change list specify its name (or tag). Multiple rule clauses can specify the same named change list. An unnamed change list appears in a rule clause. It is used only by the rule clause in which it appears.

FmtChangeListCatalog statement

The `FmtChangeListCatalog` statement defines the contents of the Format Change List Catalog. A document can have only one `FmtChangeListCatalog` statement which must appear at the top level in the order given in "MIF file layout" on page 10.

Syntax

`<FmtChangeListCatalog`	Begin Format Change List Catalog
`<FmtChangeList…>`	Defines an element (see "FmtChangeList statement," next)
`<FmtChangeList…>`	Additional statements as needed
…	
`>`	End of `FmtChangeListCatalog` statement

FmtChangeList statement

The `FmtChangeList` statement creates a format change list definition. The `FmtChangeList` statement for a named change list must appear in the `FmtChangeListCatalog` statement. The `FmtChangeList` statement for a unnamed change list must appear in the format rule clause that uses it.

A change list can specify absolute values or relative values. For example, it can specify that the paragraph left indent is one inch or it can specify that it is one inch greater than the inherited left indent. Alternatively, a change list can simply specify a paragraph catalog format to apply to a paragraph. If it does this, it can't specify changes to any other paragraph properties.

If a `FmtChangeList` statement defines a named change list, it must include an `FclTag` statement specifying its name. In addition, it must contain one statement for each paragraph format property it changes. For example, if a named change list changes only the first indent by a relative value, it contains only `FclTag` and `PgfFIndentChange` statements. If it changes the space below and the leading with absolute values, it contains `FclTag`, `PgfSpBefore`, and `PgfLeading` statements.

If a `FmtChangeList` statement changes a paragraph property to an absolute value, the statement it uses is the same as the corresponding paragraph format statement (for example, `PgfLIndent`). If the change list changes a property with a relative value, the statement it uses has the name of the corresponding paragraph format statement with the word `Change` appended to it (for example, `PgfLIndentChange`).

Syntax

Basic properties	
`<FmtChangeList`	Begin format change list
`<FclTag` *tagstring*`>`	Format change list name if the format change list is named
`<FclPgfCatalogRef` *tagstring*`>`	A paragraph catalog format to apply. If the `FmtChangeList` statement includes this statement, it can't include any of the following statements
`<PgfFIndent` *dimension*`>`	First line left margin, measured from left side of current text column
`<PgfFIndentChange` *dimension*`>`	Change to the first line left margin
`<PgfFIndentRelative` *boolean*`>`	`Yes` means the first indent is relative to the left indent instead of the left side of the current text column
`<PgfLIndent` *dimension*`>`	Left margin, measured from left side of current text column
`<PgfLIndentChange` *dimension*`>`	Change to the left margin
`<PgfRIndent` *dimension*`>`	Right margin, measured from right side of current text column
`<PgfRIndentChange` *dimension*`>`	Change to the right margin

`<PgfAlignment` *keyword*`>`	Alignment within the text column
	keyword can be one of:
	`LeftRight`
	`Left`
	`Center`
	`Right`

`<PgfSpBefore` *dimension*`>`	Space above paragraph
`<PgfSpBeforeChange` *dimension*`>`	Change to space above paragraph
`<PgfSpAfter` *dimension*`>`	Space below paragraph
`<PgfSpAfterChange` *dimension*`>`	Change to space below paragraph
`<PgfLineSpacingFixed` *boolean*`>`	`Yes` means the lines spacing is fixed (to the default font size)

<PgfLeading *dimension*>	Space below each line in a paragraph
<PgfLeadingChange *dimension*>	Change to space below each line in a paragraph
<PgfNumTabs *integer*>	Number of tabs in a paragraph. To clear all the tabs in a paragraph, specify 0
<TabStop	Begin definition of tab stop; the following property statements can appear in any order, but must appear within a TabStop statement
<TSX *dimension*>	Horizontal position of tab stop
<TSXRelative *boolean*>	Yes means the tab stop is relative to the left indent
<TSType *keyword*>	Tab stop alignment *keyword* can be one of: Left Center Right Decimal
<TSLeaderStr *string*>	Tab stop leader string (for example, `.')
<TSDecimalChar *integer*>	Align decimal tab around a character by ASCII value; in UNIX versions, type man ascii in a UNIX window for a list of characters and their corresponding ASCII values
>	End of TabStop statement
<TabStop...>	Additional statements as needed
...	
<MoveTabs *dimension*>	Move all tabs by a specified distance. A format change list can have one or more TabStob statements, or a MoveTabs statement. It can't have both
Default font name properties	
<FFamily *string*>	Name of font family
<FAngle *string*>	Name of angle
<FWeight *string*>	Name of weight
<FVar *string*>	Name of variation
<FPostScriptName *string*>	Name of font when sent to PostScript printer (see "Font name" on page 70)
<FPlatformName *string*>	Platform-specific font name, only read by the Windows version (see "FPlatformName statement" on page 71)
Default font size color and width	
<FSize *dimension*>	Size, in points only
<FSizeChange *dimension*>	Change to default font size
<FColor *tagstring*>	Font color (see "ColorCatalog statement" on page 84)
<FSeparation *integer*>	Font color; no longer used, but written out by FrameMaker for backward-compatibility (see "Color statements" on page 263)

<FStretch *percent*>	The amount to stretch or compress the font, where 100% means no change
<FStretchChange *percent*>	The amount to change the width setting for the font, where 100% means no change
Default font style	
<FUnderlining *keyword*>	Turns on underlining and specifies underlining style *keyword* can be one of: FNoUnderlining FSingle FDouble FNumeric
<FOverline *boolean*>	Turns on overline style
<FStrike *boolean*>	Turns on strikethrough style
<FChangeBar *boolean*>	Turns on the change bar
<FPosition *keyword*>	Specifies subscript and superscript characters; font size and position relative to baseline determined by Document substatements (see page 94) *keyword* can be one of: FNormal FSuperscript FSubscript
<FPairKern *boolean*>	Turns on pair kerning
<FCase *keyword*>	Applies capitalization style to string *keyword* can be one of: FAsTyped FSmallCaps FLowercase FUppercase
Default font kerning information	
<FDX *percent*>	Horizontal kern value for manual kerning expressed as percentage of an em; positive value moves characters right and negative value moves characters left
<FDY *percent*>	Vertical kern value for manual kerning expressed as percentage of an em; positive value moves characters down and negative value moves characters up
<FDW *percent*>	Spread value for space between characters expressed as percentage of an em; positive value increases the space and negative value decreases the space
<FDWChange *dimension*>	Change to spread value for space between characters expressed as percentage of an em; positive value increases the space and negative value decreases the space
Default font miscellaneous information	
<FLocked *boolean*>	Yes means the font is part of a text inset that obtains its formatting properties from the source document

Pagination properties	
<PgfPlacement *keyword*>	Vertical placement of paragraph in text column *keyword* can be one of: Anywhere ColumnTop PageTop LPageTop RPageTop
<PgfPlacementStyle *keyword*>	Placement of side heads, run-in heads, and paragraphs that straddle text columns *keyword* can be one of: Normal RunIn SideheadTop SideheadFirstBaseline SideheadLastBaseline Straddle StraddleNormalOnly
<PgfRunInDefaultPunct *string*>	Default punctuation for run-in heads
<PgfWithPrev *boolean*>	Yes keeps paragraph with previous paragraph
<PgfWithNext *boolean*>	Yes keeps paragraph with next paragraph
<PgfBlockSize *integer*>	Widow/orphan lines
Numbering properties	
<PgfAutoNum *boolean*>	Yes turns on autonumbering
<PgfNumFormat *string*>	Autonumber formatting string
<PgfNumberFont *tagstring*>	Tag from Character Catalog
<PgfNumAtEnd *boolean*>	Yes places number at end of line, instead of beginning
Advanced properties	
<PgfHyphenate *boolean*>	Yes turns on automatic hyphenation
<HyphenMaxLines *integer*>	Maximum number of consecutive lines that can end in a hyphen
<HyphenMinPrefix *integer*>	Minimum number of letters that must precede hyphen
<HyphenMinSuffix *integer*>	Minimum number of letters that must follow a hyphen
<HyphenMinWord *integer*>	Minimum length of a hyphenated word
<PgfLetterSpace *boolean*>	Spread characters to fill line
<PgfMinWordSpace *integer*>	Minimum word spacing (as a percentage of a standard space in the paragraph's default font)
<PgfOptWordSpace *integer*>	Optimum word spacing (as a percentage of a standard space in the paragraph's default font)
<PgfMaxWordSpace *integer*>	Maximum word spacing (as a percentage of a standard space in the paragraph's default font)

<PgfLanguage *keyword*>	Language to use for spelling and hyphenation *keyword* can be one of: NoLanguage USEnglish UKEnglish German SwissGerman French CanadianFrench Spanish Catalan Italian Portuguese Brazilian Danish Dutch Norwegian Nynorsk Finnish Swedish
<PgfTopSeparator *string*>	Name of reference frame (from reference page) to put above paragraph
<PgfTopSepAtIndent *boolean*>	Yes if the position of the frame specified by the PgfTopSeparator statement is at the current left indent
<PgfTopSepOffset *dimension*>	Position at which to place the reference frame above the paragraph
<PgfBotSeparator *string*>	Name of reference frame (from reference page) to put below paragraph
<PgfBotSepAtIndent *boolean*>	Yes if the position of the frame specified by the PgfBotSeparator statement is at the current left indent
<PgfBotSepOffset *dimension*>	Position at which to place the reference frame below the paragraph

Table cell properties	
<PgfCellAlignment *keyword*>	Vertical alignment for first paragraph in a cell *keyword* can be one of: Top Middle Bottom
<PgfCellLMargin *dimension*>	Left cell margin for first paragraph in a cell
<PgfCellLMarginChange *dimension*>	Change to left cell margin for first paragraph in a cell
<PgfCellBMargin *dimension*>	Bottom cell margin for first paragraph in a cell
<PgfCellBMarginChange *dimension*>	Change to bottom cell margin for first paragraph in a cell
<PgfCellTMargin *dimension*>	Top cell margin for first paragraph in a cell
<PgfCellTMarginChange *dimension*>	Change to top cell margin for first paragraph in a cell
<PgfCellRMargin *dimension*>	Right cell margin for first paragraph in a cell
<PgfCellRMarginChange *dimension*>	Change to right cell margin for first paragraph in a cell
<PgfCellLMarginFixed *boolean*>	Yes means the left cell margin is fixed
<PgfCellTMarginFixed *boolean*>	Yes means the top cell margin is fixed
<PgfCellRMarginFixed *boolean*>	Yes means the right cell margin is fixed
<PgfCellBMarginFixed *boolean*>	Yes means the bottom cell margin is fixed
>	End of FmtChangeList statement.

Elements

ElementBegin and ElementEnd statements

The ElementBegin and ElementEnd statements indicate where a structural element begins and ends. These statements must appear in a ParaLine statement (see page 182) or in a BookElements statement (see page 185).

Syntax

<ElementBegin	Begin element
<Unique *ID*>	ID, persistent across sessions, assigned when FrameMaker generates a MIF file; used by the API and should not be used by filters
<ElementReferenced *boolean*>	Yes means the element is marked as a PDF named destination for cross-references, hypertext markers, or bookmarks (version 6.0 or later)
<ETag *tagstring*>	Tag name of element from Element Catalog
<Collapsed *boolean*>	Collapse element in structure view
<SpecialCase *boolean*>	Treat element as a special case for validation
<ENamespace <	The element's namespace declarations; a declaration consists of one <ENamespacePrefix> and one <ENamespacePath>

<ENamespacePrefix *string*>	The prefix that identifies the namespace
<ENamespacePath *string*>	The system path or URI to the DTD or schema that defines the namespace
...	Additional pairs of prefix and path statements as needed
>	End of Namespace statement
<BannerTextProcessed *boolean*>	On means the banner text for the element was created once for this element. Off means the banner text was not created for this element.
<AttributeDisplay *keyword*>	Default attribute display setting for element *keyword* can be one of: AllAttributes: display all attributes ReqAndSpec: display required and specified attributes None: don't display attributes
<ElemDir *keyword*>	Direction of an element. *keyword* can be one of: LTR – Set the direction of an element to left to right. The element propagates its direction to all child elements that derive their direction from the parent element object. RTL – Set the direction of an element to right to left. The element propagates its direction to all child elements that derive their direction from the parent element object. INHERITLTR – Derive the direction from the parent object. If it resolves to left to right then INHERITLTR is assigned to ElemDir. INHERITRTL – Derive the direction from the parent object. If it resolves to right to left then INHERITRTL is assigned to ElemDir.
<Atributes	Element's attributes
<Attribute	Attribute's name and values
<AttrName *string*>	Attribute name
<AttrValue *string*>	Attribute value
<AttrValue *string*>	Attribute value if attribute allows more than one value
...	
>	End of Attribute statement
<Attribute...>	Additional statements as needed
...	
>	End of Attributes statement
<UserString *string*>	A string in which clients can store private data — can be up to 1023 characters in length
>	End of ElementBegin statement
<ElementEnd *tagstring*>	End of specified element

Usage

FrameMaker writes out the *tagstring* value in an ElementEnd statement for use by filters. Your application does not need to supply the *tagstring* value when it writes MIF files.

If the interpreter reads unbalanced ElementBegin and ElementEnd statements, it ignores superfluous element ends and closes all open elements at the end of a TextFlow statement. If the interpreter reads a flow that does not have an element enclosing all of the flow's contents, it creates a highest-level element with the tag NoName. ElementBegin and ElementEnd statements are nested within ParaLine and BookElements statements. The following example shows how FrameMaker writes an UnorderedList element:

```
<Para
    <PgfTag `Bullet'>
                # The autonumber contains a bullet and a tab.
    <PgfNumString `• \t'>
    <ParaLine
                # Note that the ElementBegin statement is nested inside both
                # the Para and ParaLine statements.
        <ElementBegin
            <ETag `UnorderedList'>
            <Collapsed No >
            <SpecialCase No >
        >       # end of ElementBegin
        <ElementBegin
            <ETag `Item'>
            <Collapsed No >
            <SpecialCase No >
        >       # end of ElementBegin
        <String `Light rail provides transportation for those who '>
    >
    <ParaLine
        <String `are unable to drive or cannot afford an automobile.'>
        <ElementEnd `Item'>
    >
>               # end of Para
<Para
        <PgfTag `Bullet'>
        <PgfNumString `• \t'>
    <ParaLine
        <ElementBegin
            <ETag `Item'>
            <Collapsed No >
            <SpecialCase No >
        >       # end of ElementBegin
        <String `Light rail lures commuters away from rush hour traffic.'>
                # Again, note that both the Item and Bulletlist elements end
                # before the end of the Para and ParaLine statements.
            <ElementEnd `Item'>
        <ElementEnd `UnorderedList'>
    >
>               # end of Para
```

PrefixEnd and SuffixBegin statements

The PrefixEnd statement appears after the ElementBegin statement and any prefix strings the element has. Everything between the ElementBegin statement and the PrefixEnd statement is treated as the element prefix. The PrefixEnd statement does not appear when the element has no prefix.

The SuffixBegin statement appears before the element suffix string, which is followed by the ElementEnd statement. Everything between the SuffixBegin statement and the ElementEnd statement is treated as the element suffix. The ElementEnd statement does not appear when the element has no suffix.

Banner text

Banner text in a FrameMaker file instructs you about what to enter in an element. Banner text is controlled using the BannerText element in the EDD. You can control the instructional text you want to display for each of the elements.

FrameMaker does not treat banner text as real content in the document. Banner text is included in FM and MIF output but is not included in XML output.

Banner text in FrameMaker is governed with the following settings:

Syntax

<BannerTextBegin >	For Internal use - please ignore
<BannerTextEnd >	For Internal use - please ignore
<EDBannerText string>	The banner text that appears inside a new element instance
<DBannerTextOn *Boolean*>	Yes turns on banner text for tags in document window.
<BannerTextProcessed *boolean*>	On means the banner text for the element was created once for this element. Off means the banner text was not created for this element.

Filter By Attribute

DefAttrValuesCatalog and AttrCondExprCatalog statements

The Filter By Attribute feature in FrameMaker supports filtering a structured document for complex output scenarios based on the value of attributes. You define a filter using a Boolean expression containing attribute-value pairs. You can create multiple filters, save them, and use them for filtering a document based on different output scenarios.

The DefAttrValuesCatalog statement and the AttrCondExprCatalog statement store information required for generating the output.

The DefAttrValuesCatalog statement defines the contents of the defined attribute values catalog. If no values are defined, the catalog is empty. Each definition has an attribute tag (AttributeTag) and a corresponding list of values (AttributeValue).

The AttrCondExprCatalog defines the contents of the filters catalog defined for a structured document. A MIF file can have only one AttrCondExprCatalog statement.

XML data for structured documents

Document and book statements

In versions 7.0 and later, FrameMaker supports XML import and export. The following statements store information necessary to properly save a document or book as XML. Statements that begin with `DXml...` are document statements, and statements that begin with `BXml...` are book statements.

Syntax

`<DXmlDocType string>` `<BXmlDocType string>`	The name given to the XML document type
`<DXmlSystemId string>` `<DXmlSystemId string>`	The system identifier for the XML document type
`<DXmlEncoding string>` `<BXmlEncoding string>`	The XML encoding parameter that was specified in the XML declaration when the XML file was opened
`<DXmlFileEncoding string>` `<BXmlFileEncoding string>`	The XML encoding that was found in the imported XML file
`<DXmlPublicId string>` `<DXmlPublicId string>`	The public identifier for the XML document type
`<DXmlStandAlone int>` `<BXmlStandAlone int>`	The XML standalone parameter that was specified in the XML declaration when the XML file was opened—determines whether or not the XML document requires a DTD
`<DXmlStyleSheet string>` `<BXmlStyleSheet string>`	The URI for the stylesheet associated with the imported XML document
`<DXmlUseBOM int>` `<BXmlUseBOM int>`	The Byte Order Mark that was specified in the imported XML document
`<DXmlWellFormed int>` `<BXmlWellFormed int>`	Indicates whether the XML document was wellformed or not
`<DXmlVersion string>` `<BXmlVersion string>`	The XML version that was specified in the XML declaration when the XML file was opened

Preference settings for structured documents

Document statement

In addition to document preferences for standard FrameMaker documents (see "Document statement" on page 88), the MIF `Document` statement describes preferences for structured FrameMaker documents.

Syntax

`<Document`	See page 88

`<DElementCatalogScope` *keyword*`>`	Validation scope
	keyword can be one of:
	`Strict`
	`Loose`
	`Children`
	`All`
	`CustomList`

`<DCustomElementList`	List of tags to display when `DElementCatalogScope` specifies `CustomList`
`<EDTag` *string*`>`	Element definition name
`<EDTag` *string*`>`	Additional statements as needed
…	
`>`	End of `DCustomElementList` statement
`<DShowElementDescriptiveTags` *boolean*`>`	`Yes` displays descriptive text against elements in the element catalog for the document.
`<DAttributeDisplay` *keyword*`>`	Default attribute display setting for document *keyword* can be one of: `AllAttributes`: display all attributes `ReqAndSpec`: display required and specified attributes `None`: don't display attributes
`<DAttrEditor` *keyword*`>`	When Edit Attributes dialog box appears for new elements *keyword* can be one of: `Never`: never `Always`: always `WhenRequired`: when there are required attributes
`<DElementBordersOn` *boolean*`>`	`Yes` turns on element borders in document window. This statement and `DElementTags` are mutually exclusive. If both statements appear in a MIF file, the later statement overrides the earlier one
`<DElementTags` *boolean*`>`	`Yes` turns on element tags in document window. This statement and `DElementBordersOn` are mutually exclusive. If both statements appear in a MIF file, the later statement overrides the earlier one
`<DBannerTextOn` *boolean*`>`	`Yes` turns on banner text for tags in document window.
`<DUseInitStructure` *boolean*`>`	`Yes` means structured FrameMaker inserts initial structure for new elements

<DUseInitStructureRecursively *boolean*>	`True` means inserting an element in a structured document will allow its child element (or elements) with their hierarchy to be inserted as defined in the EDD
<DSGMLAppName *string*>	The name of the SGML application associated with the document. For information on registering SGML applications, see the online manual *FrameMaker Structure Application Developer's Guide*
<DExclusions…>	Lists exclusions inherited when document is included in a structured book (see "ElementDef statement" on page 158)
<DInclusions…>	Lists inclusions inherited when document is included in a structured book (see "ElementDef statement" on page 158)
<DSeparateInclusions *boolean*>	`Yes` means structured FrameMaker lists inclusions separately in the element catalog
<DApplyFormatRules *boolean*>	`Yes` uses element format rules to reformat document on opening and to remove format overrides; for input filters only, not generated by FrameMaker
<DBookElementHierarchy	If the document is in a book, list of ancestors of the document's root element
<ElementContext	Describes ancestor element of the document's root element
<PrevElement	
<ETag *tagstring*>	Tag of sibling element preceding ancestor element
<Attributes …>	
>	
<Element	
<ETag *tagstring*>	Tag of ancestor element
<Attributes …>	
>	
<NextElement	
<ETag *tagstring*>	Tag of sibling element following ancestor element
<Attributes …>	
>	
>	End of `ElementContext` statement
>	End of `DBookElementHierarchy` statement
<DFCLMaximums	Upper change list limits. Format change lists cannot increment properties beyond these values
<PgfFIndent *dimension*>	Maximum first indent allowed in document
<PgfLIndent *dimension*>	Maximum left indent allowed in document
<PgfRIndent *dimension*>	Maximum right indent allowed in document
<PgfSpBefore *dimension*>	Maximum space before allowed in document

<PgfSpAfter *dimension*>	Maximum space after allowed in document
<PgfLeading *dimension*>	Maximum leading allowed in document
<FSize *dimension*>	Maximum font size allowed in document
<FDW *dimension*>	Maximum character spread allowed in document
<TSX *dimension*>	Maximum horizontal position of tab stop
<PgfCellLMargin *dimension*>	Maximum left cell margin for first paragraph in a cell
<PgfCellBMargin *dimension*>	Maximum bottom cell margin for first paragraph in a cell
<PgfCellTMargin *dimension*>	Maximum top cell margin for first paragraph in a cell
<PgfCellRMargin *dimension*>	Maximum right cell margin for first paragraph in a cell
>	End of DFCLMaximums statement
<DFCLMinimums	Lower change list limits. Format change lists cannot decrement properties below these values
<PgfFIndent *dimension*>	Minimum first indent allowed in document
<PgfLIndent *dimension*>	Minimum left indent allowed in document
<PgfRIndent *dimension*>	Minimum right indent allowed in document
<PgfSpBefore *dimension*>	Minimum space before allowed in document
<PgfSpAfter *dimension*>	Minimum space after allowed in document
<PgfLeading *dimension*>	Minimum leading allowed in document
<FSize *dimension*>	Minimum font size allowed in document.
<FDW *dimension*>	Minimum character spread allowed in document.
<TSX *dimension*>	Minimum horizontal position of tab stop
<PgfCellLMargin *dimension*>	Minimum left cell margin for first paragraph in a cell
<PgfCellBMargin *dimension*>	Minimum bottom cell margin for first paragraph in a cell
<PgfCellTMargin *dimension*>	Minimum top cell margin for first paragraph in a cell
<PgfCellRMargin *dimension*>	Minimum right cell margin for first paragraph in a cell
>	End of DFCLMinimums statement
<WEBDAV	
<DocServerUrl *string*>	URL of the MIF file on the WEBDAV Server. Any HTTP path is valid. Example: <DocServerUrl `http://mikej-xp/joewebdav/myfile.mif'> #http://mikej-xp/joewebdav is the path of the server.

<DocServerState>	Valid values: • CheckedOut if checked out • CheckedIn if not checked out
>	End of WEBDAV Document statement
>	End of Document statement

Text in structured documents

TextLine statement

Text lines cannot contain elements.

ParaLine statement

The ParaLine statement defines a line within a paragraph. It must appear in a Para statement.

Syntax

<ParaLine	
<ElementBegin...>	Begin structural element (see page 174)
<ElementEnd *tagstring*>	End structural element
>	End of ParaLine statement

Usage

A typical ParaLine statement consists of one or more String, Char, ATbl, AFrame, FNote, Variable, XRef, and Marker statements that define the contents of the line of text. These statements are interspersed with statements that indicate the scope of document components such as structural elements and conditional text.

Structured book statements

A structured book file contains documents that were created in FrameMaker. These documents normally contain structural elements. A structured book file has the same book statements that appear in a normal book file plus two additional types of information about structural elements:

- An Element Catalog defined in ElementDefCatalog
- A structure tree defined in BookElements

ElementDefCatalog statement

The ElementDefCatalog statement contains the definitions of all elements in the book file. A book file can have only one ElementDefCatalog statement. It normally appears near the beginning of the file.

Syntax

`<ElementDefCatalog`	Begin Element Catalog
`<ElementDef…>`	Element definitions (defined on page 158)
`<ElementDef…>`	Additional statements as needed
…	
`>`	End of `ElementDefCatalog` statement

Usage

The book file inherits the Element Catalog from the document used to generate the book file or from a document given as the source for the Import>Element Definitions command. In a MIF file, you should copy the Element Catalog from one of the structure documents included in the book.

BookSettings statement

The `BookSettings` statement contains the definitions of all elements in the book file. A book file can have only one `BookSettings` statement. It normally appears near the beginning of the file. The statements in the `BookSettings` statement correspond to statements in the `BookSettings` statement, except that they begin with the letter `B` instead of the letter `D`.

Syntax

`<BookSettings`	Begin book settings

`<BElementCatalogScope` *keyword*`>`	Validation scope
	keyword can be one of:
	`Strict`
	`Loose`
	`Children`
	`All`
	`CustomList`

`<BCustomElementList`	List of tags to display when `BElementCatalogScope` specifies `CustomList`
`<EDTag` *string*`>`	Element definition name
`<EDTag` *string*`>`	Additional statements as needed
…	
`>`	End of `DCustomElementList` statement
`<BShowElementDescriptiveTags` *boolean*`>`	`Yes` displays descriptive text against elements in the element catalog for the book.
`<BAttributeDisplay` *keyword*`>`	Default attribute display setting for document *keyword* can be one of: `AllAttributes`: display all attributes `ReqAndSpec`: display required and specified attributes `None`: don't display attributes

<BAttrEditor *keyword*>	When Edit Attributes dialog box appears for new elements *keyword* can be one of: Never: never Always: always WhenRequired: when it is required
<BUseInitStructure *boolean*>	Yes means structured FrameMaker inserts initial structure for new elements
<BUseInitStructureRecursively *boolean*>	True means inserting an element in a structured book will allow its child element (or elements) with their hierarchy to be inserted as defined in the EDD
<BSGMLAppName *string*>	The name of the SGML application associated with the document. For information on registering SGML applications, see the online manual *FrameMaker Structure Application Developer's Guide*
<BSeparateInclusions *boolean*>	Yes means structured FrameMaker lists inclusions separately in the element catalog
<BFCLMaximums	Upper change list limits. Format change lists cannot increment properties beyond these values
<PgfFIndent *dimension*>	Maximum first indent allowed in book
<PgfLIndent *dimension*>	Maximum left indent allowed in book
<PgfRIndent *dimension*>	Maximum right indent allowed in book
<PgfSpBefore *dimension*>	Maximum space before allowed in book
<PgfSpAfter *dimension*>	Maximum space after allowed in book
<PgfLeading *dimension*>	Maximum leading allowed in book
<FSize *dimension*>	Maximum font size allowed in book
<FDW *dimension*>	Maximum character spread allowed in book
<TSX *dimension*>	Minimum horizontal position of tab stop
<PgfCellLMargin *dimension*>	Minimum left cell margin for first paragraph in a cell
<PgfCellBMargin *dimension*>	Minimum bottom cell margin for first paragraph in a cell
<PgfCellTMargin *dimension*>	Minimum top cell margin for first paragraph in a cell
<PgfCellRMargin *dimension*>	Minimum right cell margin for first paragraph in a cell
>	End of BFCLMaximums statement
<BFCLMinimums	Lower change list limits. Format change lists cannot decrement properties below these values
<PgfFIndent *dimension*>	Minimum first indent allowed in book
<PgfLIndent *dimension*>	Minimum left indent allowed in book
<PgfRIndent *dimension*>	Minimum right indent allowed in book
<PgfSpBefore *dimension*>	Minimum space before allowed in book
<PgfSpAfter *dimension*>	Minimum space after allowed in book

<PgfLeading *dimension*>	Minimum leading allowed in book
<FSize *dimension*>	Minimum font size allowed in book
<FDW *dimension*>	Minimum character spread allowed in book
<TSX *dimension*>	Minimum horizontal position of tab stop
<PgfCellLMargin *dimension*>	Minimum left cell margin for first paragraph in a cell
<PgfCellBMargin *dimension*>	Minimum bottom cell margin for first paragraph in a cell
<PgfCellTMargin *dimension*>	Minimum top cell margin for first paragraph in a cell
<PgfCellRMargin *dimension*>	Minimum right cell margin for first paragraph in a cell
>	End of BFCLMinimums statement
>	End of BookSettings statement

BookElements statement

The BookElements statement contains all of the elements in the book's hierarchy. This statement must appear after the BookComponent statements. Otherwise, the MIF interpreter warns you about out-of-bounds EComponent values.

Syntax

<BookElements	Begin structure tree
<ElementBegin…>	Begin element that contains other elements
<ElementEnd>	End element that contains other elements
<ElementBegin…>	Additional statements as needed
<ElementEnd>	
<Element	Begin element with no subelements
<ETag *tagstring*>	Element tag name from Element Catalog
<EComponent *integer*>	Corresponding book component (numbering starts at 1)
<ETextSnippet *string*>	Text snippet for structure window
>	End of Element statement
<Element…>	Additional statements as needed
>	End of BookElements statement

Usage

The ElementBegin and ElementEnd statements define elements that contain other elements.

The Element statement defines an element with no subelements. If the element is inserted in the book structure from the Element Catalog, this statement includes only the ETag substatement. If the element corresponds to a book component, this statement encodes the sequence number of the corresponding component file. If the element corresponds to an unstructured component file, the ETag string value is empty. (For more information about structured documents, see *Using FrameMaker*.)

MIF Messages

Invalid context specification: parameter.
There is a syntax error in an <EDContextSpec> statement in an element definition.

EDContainerType has an invalid value.
An <EDContainerType> statement uses an invalid value.

EDContainerType ignored for object element definition.
An element definition contains an <EDContainerType> statement but the <ObjectType> statement doesn't specify EDContainer.

Value of EDObject is invalid.
An <EDObject> statement uses an invalid value.

General rule not allowed for object element definition.
An element definition for an object element contains an <EDGeneralRule> statement.

Exclusions not allowed for object element definition.
An element definition for an object element contains an <EDExclusions> statement.

Inclusions not allowed for object element definition.
An element definition for an object element contains an <EDInclusions> statement.

Discarding element definition--no EDTag name was specified.
An element definition has no tag name, so it is ignored.

Bad general rule for element definition: Name or '(' expected.
A general rule is invalid.

Bad general rule for: Cannot use different connectors in a group.
A general rule is invalid.

Bad general rule for: '(' expected.
A general rule is invalid.

Bad general rule for element definition: ')' expected.
A general rule is invalid.

Ambiguous general rule for element definition:
A general rule is invalid.

Bad general rule for element definition: Syntax Error.
A general rule is invalid.

Bad general rule for element definition: Connector (, or | or &) expected.
A general rule is invalid.

Duplicate definition: only first element definition for tag will be used.
Two or more element definitions use the same tag.

Format tag is invalid for an element of type EDEquation - defaulting to Medium.
Only small, medium, and large format tags are valid for an equation element.

Element name contains characters that are not allowed.
Element name contains at least one disallowed character, such as `&`, `|`, or `*`.

Chapter 6: MIF Equation Statements

This chapter describes the MIF statements that define equations. Use it as a reference when you write filters for translating documents that include equations. For more information about creating and editing equations, see your Adobe® FrameMaker® user's manual.

MathML statement

FrameMaker provides support for MathML, which is an XML application for representing mathematical notation. This support is provided through out-of-the-box integration with MathFlow Editor by Design Science. FrameMaker includes 30-day trial licenses of the following MathFlow editors: Style Editor and Structure Editor.

Following is a sample MIF tags snippet that shows MathML MIF syntax:

```
<MathML
<MathMLDataLen 489>
   <MathMLData `'>
   <ShapeRect  0.0" 0.0" 1.30666" 0.59999">
   <BRect  0.0" 0.0" 1.30666" 0.59999">
   <MathMLDpi 150>
   <MathMLComposeDpi 300>
   <MathMLfontSize 14>
   <MathMLinLine Yes>
   <MathMLApplyPgfStyle Yes>
   <MathMLFlipLR No>
  > # end of MathML
```

Syntax

<MathML	
<MathMLDataLen *integer*>	Number of characters in the equation's XML.
<MathMLData *String*>	The actual data of XML representation of the MathML equation. Using the XML tags, the MIF file displays the structure of the equation. For example: <math\>\x0d <mrow\>\x0d <msqrt\>\x0d <mrow\>\x0d <msup\>\x0d <mrow\>\x0d <mi\>a</mi\>\x0d </mrow\>\x0d <mrow\>\x0d <mn\>2</mn\>\x0d </mrow\>\x0d </msup\>\x0d <mo\>+</mo\>\x0d <msup\>\x0d <mrow\>\x0d <mi\>b</mi\>\x0d </mrow\>\x0d <mrow\>\x0d <mn\>2</mn\>\x0d </mrow\>\x0d </msup\>\x0d </mrow\>\x0d </msqrt\>\x0d </mrow\>\x0d </math\>\x0d '> <ShapeRect 0.0" 0.0" 1.30666" 0.59999"
<MathMLDpi *integer*>	Scaling value for the image file created for the equation.
<MathMLComposeDpi *integer*>	To show the equation corresponding to MathML FrameMaker creates a temporary image and this ComposeDpi is used to provide the resolution at the time of creation of that image.
<MathMLfontSize *integer*>	The font size of the MathML equation content.
<MathMLinLine *boolean*>	Yes places the equation inline with the enclosing paragraph.

<MathMLApplyPgfStyle boolean>	Yes applies the formats of the enclosing paragraph to the equation. Formats include, the font, font family, background color, and foreground color.
<MathMLFlipLR *boolean*>	Yes inverts the equation image sideways.

Document statement

In addition to document preferences (see "Document statement" on page 88), the MIF Document statement describes standard formats for equations. The equation formatting substatements correspond to settings in the Equations palette.

Syntax

<Document	See "Document statement" on page 88
Equation sizes	
<DMathSmallIntegral *dimension*>	Size in points of integral symbols in small equations
<DMathMediumIntegral *dimension*>	Size in points of integral symbols in medium equations
<DMathLargeIntegral *dimension*>	Size in points of integral symbols in large equations
<DMathSmallSigma *dimension*>	Size in points of summation and product symbols in small equations
<DMathMediumSigma *dimension*>	Size in points of summation and product symbols in medium equations
<DMathLargeSigma *dimension*>	Size in points of summation and product symbols in large equations
<DMathSmallLevel1 *dimension*>	Size in points of level 1 expression (normal level) in small equations
<DMathMediumLevel1 *dimension*>	Size in points of level 1 expression in medium equations
<DMathLargeLevel1 *dimension*>	Size in points of level 1 expression in large equations
<DMathSmallLevel2 *dimension*>	Size in points of level 2 expression (first level subscripts and superscripts) in small equations
<DMathMediumLevel2 *dimension*>	Size in points of level 2 expression in medium equations
<DMathLargeLevel2 *dimension*>	Size in points of level 2 expression in large equations
<DMathSmallLevel3 *dimension*>	Size in points of level 3 expression (second level subscripts and superscripts) in small equations
<DMathMediumLevel3 *dimension*>	Size in points of level 3 expression in medium equations
<DMathLargeLevel3 *dimension*>	Size in points of level 3 expression in large equations
<DMathSmallHoriz *integer*>	Horizontal spread for small equations expressed as a percentage of equation's point size; negative values decrease space and positive values increase space
<DMathMediumHoriz *integer*>	Horizontal spread for medium equations
<DMathLargeHoriz *integer*>	Horizontal spread for large equations

<DMathSmallVert *integer*>	Vertical spread for small equations expressed as a percentage of equation's point size; negative values decrease space and positive values increase space
<DMathMediumVert *integer*>	Vertical spread for medium equations
<DMathLargeVert *integer*>	Vertical spread for large equations
<DMathShowCustom *boolean*>	Specifies whether to show all math elements or only custom elements in Insert Math Element dialog box
<DMathFunctions *tagstring*>	Font for functions
<DMathNumbers *tagstring*>	Font for numbers
<DMathVariables *tagstring*>	Font for variables
<DMathStrings *tagstring*>	Font for strings
<DMathGreek *tagstring*>	Font for Greek characters
<DMathCatalog…>	Describes custom math elements (see "DMathCatalog statement," next)
>	End of `Document` statement

DMathCatalog statement

The `DMathCatalog` statement describes the custom math elements in a document. It must appear in a `Document` statement.

Syntax

<DMathCatalog	Lists custom math elements
<DMathGreekOverrides *tagstring*>	Identifies a redefined Greek symbol and forces lookup on reference page; *tagstring* argument must match the name of reference frame
<DMathGreekOverrides *tagstring*>	Additional statements as needed
…	
<DMathOpOverrides	Identifies built-in operator with redefined display properties
<DMathOpName *tagstring*>	Name of built-in operator from reference frame
<DMathOpTLineOverride *boolean*>	`No` uses default glyph for operator; `Yes` looks up operator on text line in reference frame
<DMathOpPositionA *integer*>	Position of first operand expressed as a percentage of equation font size
<DMathOpPositionB *integer*>	Position of second operand
<DMathOpPositionC *integer*>	Position of third operand
>	End of `DMathOpOverrides` statement
<DMathNew	Defines new math element
<DMathOpName *tagstring*>	Name of math element from reference frame

<DMathNewType *keyword*>	Specifies custom math element type; for a list of types, see the chapter on creating equations in your user's manual *keyword* can be one of: Atom Delimiter Function Infix Large Limit Postfix Prefix VerticalList
<DMathOpTLineOverride *boolean*>	No uses default glyph for operator; Yes looks up operator on text line in reference frame
<DMathOpPositionA *integer*>	Position of first operand expressed as a percentage of equation font size
<DMathOpPositionB *integer*>	Position of second operand
<DMathOpPositionC *integer*>	Position of third operand
>	End of DMathNew statement
>	End of DMathCatalog statement

Usage

You can define new math elements or redefine math elements that appear on the Equations palette. To create a custom math element, add the element's name and type to the DMathCatalog statement. On a reference page with a name beginning with the word *FrameMath*, define the math element in a named unanchored graphic frame. In the frame (called a reference frame), create a text line that contains one or more characters that represent the math symbol; you can apply specialized math fonts and change the position of the characters to get the appearance you want. You can use custom elements in equations by including them in a MathFullForm statement.

For example, to create a symbol for the set of real numbers, add the new element to the Math Catalog as follows:

```
<Document
<DMathCatalog
        <DMathNew
                # Name of new math element
            <DMathOpName `Real Numbers'>
                # Type of math element
            <DMathNewType Atom >
        >       # end of DMathNew
    >           # end of DMathCatalog
>               # end of Document
```

Define the custom element on a reference page that has a name beginning with *FrameMath*:

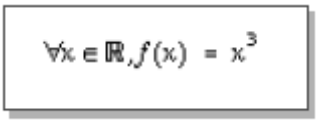
```
<Page
                # Create a named reference page.
    <PageType ReferencePage >
    <PageTag `FrameMath1'>
                # Create a named, unanchored frame.
    <Frame
        <FrameType NotAnchored >
        <Tag `Real Numbers'>
        ...
                # Create the math element in the first text line in the frame.
        <TextLine
```

```
        ...
                # Apply a specialized math font to the letter R.
            <Font
                <FTag `'>
                <FFamily `MathematicalPi'>
                <FVar `Six'>
                <FWeight `Regular'>
            >   # end of Font
            <String `R'>
        >       # end of TextLine
    >           # end of Frame
>               # end of Page
```

To insert the new element in an equation, use the `char` expression (see page 197) and the element's name in a `MathFullForm` statement as shown in the following equation:

```
<MathFullForm `equal[in[forall[char[x]], comma[char[(*T"Real Numbers"T*)New],
times[char[f],id[char[x]]]]], indexes[1,0,char[x],num[3.00000000,"3"]]]'
>               # end of MathFullForm
```

The equation looks like this in the FrameMaker document:



You can change the appearance of a built-in math element, although you cannot change the element's type or behavior. For example, to redefine the built-in inverse sine function (*asin*) so that it appears as sin^{-1}, add the redefined element to the Math Catalog as follows:

```
<DMathCatalog
    <DMathOpOverrides
                # The name of the built-in operator as it appears in MIF.
        <DMathOpName `asin'>
                # Forces lookup from the reference page.
        <DMathOpTLineOverride Yes >
    >           # end of DMathOpOverrides
>               # end of DMathCatalog
```

Redefine the appearance of the element in a reference frame as follows:

```
<Page
                # Create a named reference page.
    <PageType ReferencePage >
    <PageTag `FrameMath1'>
                # Create a named, unanchored frame.
    <Frame
        <FrameType NotAnchored >
        ...
                # The name of the built-in element as it appears in
                # the Equations palette.
        <Tag `Inverse Sine'>
                # Define the element in the first text line in the frame.
        <TextLine
            ...
                # Apply a new font style and position to change the
                # appearance of the math element.
            <Font
                <FTag `'>
                <FWeight `Regular'>
            >   # end of Font
            <String `sin'>
            <Font
```

```
                <FTag `'>
                <FWeight `Regular'>
                <FPosition FSuperscript >
            >   # end of Font
            <String `-1 '>
        >       # end of TextLine
    >           # end of Frame
>               # end of Page
```

When you create the reference frame that specifies the new appearance of the math element, you must give the frame the name of the built-in element as it appears in the Equations palette. To find the name of a built-in element, choose Insert Math Element from the equations pop-up menu on the Equations palette. Turn off Show Custom Only in the dialog box and scroll through the element names until you find the one you want.

To use the redefined element in an equation, include the `asin` expression (see page 202) along with the name of the reference frame as follows:

```
<MathFullForm `asin[(*T"Inverse Sine"T*)char[x]]'
>           # end of MathFullForm
```

For more information about including custom operators in equations, see "Custom operators" on page 211. For more information about format codes, see "MathFullForm statement syntax" on page 195.

Math statement

A `Math` statement describes an equation within a document. It can appear at the top level or within a `Page` or `Frame` statement.

Syntax

<Math	
Generic object statements	Information common to all objects (see "Generic object statements" on page 112)
<Angle *integer*>	Angle of rotation in degrees: `0`, `90`, `180`, `270`
<ShapeRect *L T W H*>	Position and size of bounding rectangle, before rotation, in enclosing page or frame
<MathFullForm *string*>	Description of equation (defined in "MathFullForm statement syntax" on page 195)
<MathLineBreak *dimension*>	Allows automatic line breaks after this position
<MathOrigin *X Y*>	Position of equation in current frame or page

<MathAlignment *keyword*>	Alignment of equation within `ShapeRect`
	keyword can be one of:
	`Left`
	`Center`
	`Right`
	`Manual`

<MathSize *keyword*>	Equation size (defined on page 189)
	keyword can be one of:
	`MathLarge`
	`MathMedium`
	`MathSmall`

>	End of `Math` statement

Usage

Values of the `ShapeRect` statement specify the coordinates and size of the bounding rectangle before it is rotated. The equation is rotated by the value specified in an `Angle` statement. The `MathFullForm` string defines the mathematical properties of the equation. For a complete description, see "MathFullForm statement," next.

Whenever you save a document as a MIF file using the Save As command, FrameMaker writes all the `Math` substatements, except `ObColor`, to the file. It writes an `ObColor` statement only when the equation is in a color other than black. The `ObColor` statement specifies the color for the entire equation object. To specify color for an individual element within an equation, use the formatting code `(*q`*string*`q*)` (see "MathFullForm statement syntax" on page 195).

If you are writing an output filter for converting FrameMaker equations to a format used by another application, you might be able to ignore some of the `Math` substatements. You don't need MIF statements for FrameMaker's math features that are unsupported by another application.

If you are writing an input filter for converting equations created with another application to FrameMaker equations, you must provide a `ShapeRect` or `MathOrigin` substatement to specify the equation's location on the page. The other `Math` substatements are not required. If you don't provide them, the MIF interpreter uses preset values. If you don't define the equation in a `MathFullForm` statement, an equation prompt appears in the FrameMaker document.

MathFullForm statement

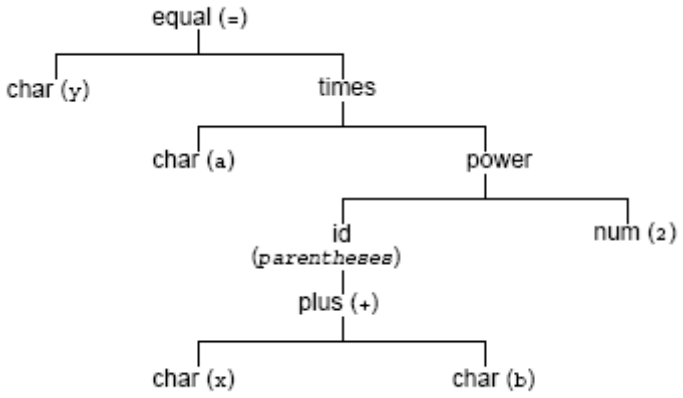
The `MathFullForm` statement consists of a string containing a series of expressions that define the mathematical structure of an equation. Each expression defines a component of the equation and can be nested within other expressions.

A sample MathFullForm statement

This example shows an equation and the `MathFullForm` statement that defines it. The diagram shows the hierarchy of the `MathFullForm` statement. Symbols that appear in the equation are shown in parentheses following the `MathFullForm` expression.

$$y = a(x+b)^2$$

```
<MathFullForm `equal[char[y],times[char[a],power[id[plus[char[x],char[b]]],num[2,"2"]]]]'
>           # end of MathFullForm
```



MathFullForm statement syntax

In addition to the mathematical structure of the equation, a `MathFullForm` statement can contain special instructions for character formatting, manual alignment points, and positioning and spacing values. Expressions have the following syntax:

ExpressionName`[(*`*FormatCodes*`*)`*operand*`,`*operand*`,...]`

Where	Is
ExpressionName	The expression name (for example, `abs`)
FormatCodes	Optional formatting codes (for example, `i2i`), described next
operand	Another expression

Formatting codes are enclosed within asterisk (`*`) delimiters. If an expression doesn't contain formatting codes, it cannot contain asterisks. Formatting codes consist of a pair of flags enclosing a numeric value or string, except for boolean flags, which are a single flag. For example, the following expression contains formatting codes that select a display format and a boolean flag to set a manual line break point:

```
<MathFullForm `id[(*i2i*)char[x]]'>
```

String values in format codes must be enclosed in straight, double quotation marks ("). To include characters in the extended ASCII range (above `0x127`), use a backslash sequence (see "Character set in strings" on page 7).

You can use the following formatting codes, which can appear in any order. The default for all numeric values is 0.

Format code	Meaning
A*integer*A	Manual alignment mark in element (0=none, 1=right, 2=left)
b*metric*b	Extra space at bottom of expression; corresponds to Spacing values in the Position Settings dialog box
B*string*B	Font angle (for example, "`Italic`")
c*integer*c	Alignment for horizontal lists and matrices (0=baseline, 1=top, 2=bottom)
C*integer*C	Character case
D*integer*D	Double underline (0=no underline, 1=underline)

Format code	Meaning
f*string*f	Font family (for example, f"Times"f)
i*integer*i	Display format number (0, 1, 2)
j*integer*j	Alignment for vertical lists and matrices (0=center, 1=left, 2=right,3=at equal symbol, 4=left of equal symbol)
l*metric*l	Extra space to left of expression; corresponds to Spacing values in the Position Settings dialog box
M	In a matrix, makes all column widths equal (boolean)
m	In a matrix, makes all row heights equal (boolean)
n	No automatic parentheses (boolean)
N*integer*N	Numeric underline (0=no underline, 1=underline)
o*integer*o	Outline (0=no outline, 1=outline)
O*integer*O	Overline (0=no overline, 1=overline)
q*string*q	Color name (for example, "Red")
r*metric*r	Extra space to right of expression; corresponds to Spacing values in the Position Settings dialog box
R*integer*R	Shadow (0=no shadow, 1=shadow)
s*decimal*s	Character size in points (for example, s12.00s)
S*integer*S	Strikeout (0=no strikeout, 1=strikeout)
t*metric*t	Extra space at top of expression; corresponds to Spacing values in the Position Settings dialog box
T*string*T	Name of custom element from reference page frame
u	Manual line break to left (boolean)
U*integer*U	Underline (0=no underline, 1=underline)
v	Manual line break to right (boolean)
V*string*V	Font variation (for example, "Narrow")
W*string*W	Font weight (for example, "Bold")
x*metric*x	Horizontal kern value
y*metric*y	Vertical kern value

When expressions have multiple display formats, there is one default format. Additional formats are numbered. For example, the id expression has three display formats.

Example	MathFullForm statement
(x)	<MathFullForm `id[char[x]]'>
[x]	<MathFullForm `id[(*i1i*)char[x]]'>
{x}	<MathFullForm `id[(*i2i*)char[x]]'>

Atomic expressions

Atomic expressions are expressions that don't take other expressions as operands. They usually act as operands in more complex expressions.

prompt

`prompt` is a placeholder to show an expression's undefined operands. Of the character formatting specifications, only kerning values affect the appearance of a prompt.

Example	MathFullForm statement
?	`<MathFullForm `prompt[]'>`

num

`num` describes a number. It always has two operands: the first shows the number as used for computations (internal precision), and the second shows the number as displayed. When fewer digits are displayed than are used internally, an ellipsis appears after the number.

Example	MathFullForm statement
3.1415927	`<MathFullForm `num[3.141592653589793,"3.1415927"]'>`

There are two special cases of the `num` expression.

Example	MathFullForm statement
`Infinity`	`<MathFullForm `num[Infinity,"Infinity"]'>`
`NaN`	`<MathFullForm `num[NaN,"NaN"]'>`

`NaN` means not a number. These forms of `num` usually result from computations.

string

`string` contains a character string. Character strings must be enclosed in straight, double quotation marks ("). To include characters in the extended ASCII range (above `0x127`), use a backslash sequence (see "Character set in strings" on page 7). To include a straight, double quotation mark, precede the quotation mark with a straight, double quotation mark.

Example	MathFullForm statement
`FrameMath`	`<MathFullForm `string["FrameMath"]'>`
`using "quotes"`	`<MathFullForm `string["using ""quotes"""]'>`

char

`char` describes a character.

Example	MathFullForm statement
x	`<MathFullForm `char[x]'>`

The `char` expression can contain one of the letters *a* through *z*, one of the letters *A* through *Z*, a custom math element, or one of the character names shown in the following table.

Example	MathFullForm statement
ℵ	`<MathFullForm `char[aleph]'>`
α	`<MathFullForm `char[alpha]'>`
β	`<MathFullForm `char[beta]'>`
⊥	`<MathFullForm `char[bot]'>`
χ	`<MathFullForm `char[chi]'>`
∂	`<MathFullForm `char[cpartial]'>`
∘	`<MathFullForm `char[degree]'>`
δ	`<MathFullForm `char[delta]'>`
Δ	`<MathFullForm `char[Delta]'>`
∅	`<MathFullForm `char[emptyset]'>`
ε	`<MathFullForm `char[epsilon]'>`
η	`<MathFullForm `char[eta]'>`
γ	`<MathFullForm `char[gamma]'>`
Γ	`<MathFullForm `char[Gamma]'>`
ℑ	`<MathFullForm `char[Im]'>`
∞	`<MathFullForm `char[infty]'>`
ι	`<MathFullForm `char[iota]'>`
κ	`<MathFullForm `char[kappa]'>`
λ	`<MathFullForm `char[lambda]'>`
Λ	`<MathFullForm `char[Lambda]'>`
…	`<MathFullForm `char[ldots]'>`
μ	`<MathFullForm `char[mu]'>`
∇	`<MathFullForm `char[nabla]'>`
ν	`<MathFullForm `char[nu]'>`

Example	MathFullForm statement
ω	<MathFullForm `char[omega]'>
Ω	<MathFullForm `char[Omega]'>
ϕ	<MathFullForm `char[phi]'>
Φ	<MathFullForm `char[Phi]'>
π	<MathFullForm `char[pi]'>
Π	<MathFullForm `char[Pi]'>
″	<MathFullForm `char[pprime]'>
′	<MathFullForm `char[prime]'>
ψ	<MathFullForm `char[psi]'>
Ψ	<MathFullForm `char[Psi]'>
ℜ	<MathFullForm `char[Re]'>
ρ	<MathFullForm `char[rho]'>
σ	<MathFullForm `char[sigma]'>
Σ	<MathFullForm `char[Sigma]'>
τ	<MathFullForm `char[tau]'>
θ	<MathFullForm `char[theta]'>
Θ	<MathFullForm `char[Theta]'>
υ	<MathFullForm `char[upsilon]'>
ϒ	<MathFullForm `char[Upsilon]'>
φ	<MathFullForm `char[varphi]'>
ϖ	<MathFullForm `char[varpi]'>
ς	<MathFullForm `char[varsigma]'>
ϑ	<MathFullForm `char[vartheta]'>
℘	<MathFullForm `char[wp]'>
ξ	<MathFullForm `char[xi]'>

Example	MathFullForm statement
Ξ	`<MathFullForm `char[Xi]'>`
ζ	`<MathFullForm `char[zeta]'>`

Using char for custom math elements

The `char` expression can contain a custom math element by using the following syntax:

`<MathFullForm `char[(*T"`*ElementName*`"T*)New]'>`

where *ElementName* is the name of the reference frame that contains the custom element.

Using char and diacritical for diacritical marks

`The char and the diacritical` expressions both describe diacritical marks around an operand.

The char expression places diacritical marks around a single operand, as shown in the following table. The `char` expression is backward-compatible.

Example	<MathFullForm> statement
$\grave{x}$	`<MathFullForm `char[x,1,0,0,0,0]'>`
$\hat{x}$	`<MathFullForm `char[x,2,0,0,0,0]'>`
x	`<MathFullForm `char[x,3,0,0,0,0]'>`
x'	`<MathFullForm `char[x,0,1,0,0,0]'>`
x''	`<MathFullForm `char[x,0,2,0,0,0]'>`
x'''	`<MathFullForm `char[x,0,3,0,0,0]'>`
$\dot{x}$	`<MathFullForm `char[x,0,0,1,0,0]'>`
$\ddot{x}$	`<MathFullForm `char[x,0,0,2,0,0]'>`
$\dddot{x}$	`<MathFullForm `char[x,0,0,3,0,0]'>`
$\bar{x}$	`<MathFullForm `char[x,0,0,0,1,0]'>`
$\underset{\sim}{x}$	`<MathFullForm `char[x,0,0,0,0,1]'>`
$\underline{x}$	`<MathFullForm `char[x,0,0,0,0,2]'>`

The `char` expression can also describe composite diacritical marks. The following table contains examples.

Example	MathFullForm statement
$\underline{\grave{x}}$	`<MathFullForm `char[x,1,0,0,0,2]'>`
$\underline{x}'$	`<MathFullForm `char[x,3,1,0,0,2]'>`

The `diacritical` expression places diacritical marks around multiple operands and describes two additional diacritical marks. The `diacritical` expression describes the same marks that the `char` expression describes, but it can take multiple operands. In addition, the `diacritical` expression describes two forms of diacritical mark not described by the `char` expression. The following table shows examples of `diacritical` expressions.

Example	MathFullForm statement
$\vec{x}$	`<MathFullForm `diacritical[4,0,0,0,0,char[x]]'>`
$\widehat{x}$	`<MathFullForm `diacritical[5,0,0,0,0,char[x]]'>`
$\overrightarrow{AB}$	`<MathFullForm` `` `diacritical[4,0,0,0,0,times[char[A],char[B]]]'> ``

Note: *The* `diacritical` *expression is not backward compatible. When an earlier version (previous to 4.x) of FrameMaker reads a MIF file saved in version 4 or later of FrameMaker, any equations that contain* `diacritical` *expressions are lost. You should edit any* `MathFullForm` *statements that contain* `diacritical` *expressions before opening the file in earlier versions of FrameMaker. For more information, see* *"Math statements" on page 264*.

dummy

The `dummy` expression describes a dummy variable that you can use as a placeholder in equations. For example, in the following equation, *i* is a dummy variable:

$$\sum_{i=0}^{4} x^{i} = 1 + x + x^{2} + x^{3} + x^{4}$$

The `dummy` expression has the same syntax as the `char` expression and can contain the same character symbols or names.

Example	MathFullForm statement
x	`<MathFullForm `dummy[x]'>`

Operator expressions

Operator expressions take at least one expression as an operand. There are no restrictions on the complexity of operator expressions, and they are not restricted by any concepts of domain or typing.

Unary operators

Unary operators have one expression as an operand. Three of the unary operators—`id`, `lparen`, and `rparen`—have multiple display formats. The following table contains an example of each unary operator (in all of its display formats) with `char[x]` as a sample operand.

Example	MathFullForm statement
$\lvert x \rvert$	`<MathFullForm `abs[char[x]]'>`
$\operatorname{acos} x$	`<MathFullForm `acos[char[x]]'>`
$\operatorname{acosh} x$	`<MathFullForm `acosh[char[x]]'>`

Example	MathFullForm statement
$\mathrm{acot}x$	<MathFullForm `acot[char[x]]'>
$\mathrm{acoth}x$	<MathFullForm `acoth[char[x]]'>
$\mathrm{acsc}x$	<MathFullForm `acsc[char[x]]'>
$\mathrm{acsch}x$	<MathFullForm `acsch[char[x]]'>
$\angle x$	<MathFullForm `angle[char[x]]'>
$\mathrm{arg}x$	<MathFullForm `arg[char[x]]'>
$\mathrm{asec}x$	<MathFullForm `asec[char[x]]'>
$\mathrm{asech}x$	<MathFullForm `asech[char[x]]'>
$\mathrm{asin}x$	<MathFullForm `asin[char[x]]'>
$\mathrm{asinh}x$	<MathFullForm `asinh[char[x]]'>
x^*	<MathFullForm `ast[char[x]]'>
$\mathrm{atan}x$	<MathFullForm `atan[char[x]]'>
$\mathrm{atanh}x$	<MathFullForm `atanh[char[x]]'>
$\Box x$	<MathFullForm `box[char[x]]'>
$\Box^2 x$	<MathFullForm `box2[char[x]]'>
$\Box \bullet x$	<MathFullForm `boxdot[char[x]]'>
$\langle x\vert$	<MathFullForm `bra[char[x]]'>
$\lceil x \rceil$	<MathFullForm `ceil[char[x]]'>
Δx	<MathFullForm `change[char[x]]'>
$\cos x$	<MathFullForm `cos[char[x]]'>
$\cosh x$	<MathFullForm `cosh[char[x]]'>
$\cot x$	<MathFullForm `cot[char[x]]'>
$\coth x$	<MathFullForm `coth[char[x]]'>
$\csc x$	<MathFullForm `csc[char[x]]'>
$\mathrm{csch}x$	<MathFullForm `csch[char[x]]'>

Example	MathFullForm statement
$\nabla\times x$	<MathFullForm `curl[char[x]]'>
$x^{\dagger}$	<MathFullForm `dagger[char[x]]'>
$\langle x\rangle$	<MathFullForm `dangle[char[x]]'>
dx	<MathFullForm `diff[char[x]]'>
$\nabla\bullet x$	<MathFullForm `diver[char[x]]'>
$\underbrace{x}$	<MathFullForm `downbrace[char[x]]'>
$\exp x$	<MathFullForm `exp[char[x]]'>
$\exists x$	<MathFullForm `exists[char[x]]'>
$x!$	<MathFullForm `fact[char[x]]'>
$\lfloor x \rfloor$	<MathFullForm `floor[char[x]]'>
$\forall x$	<MathFullForm `forall[char[x]]'>
(x)	<MathFullForm `id[char[x]]'>
$[x]$	<MathFullForm `id[(*i1i*)char[x]]'>
$\{x\}$	<MathFullForm `id[(*i2i*)char[x]]'>
$\mathrm{imag}\,x$	<MathFullForm `imag[char[x]]'>
$\|x\rangle$	<MathFullForm `ket[char[x]]'>
$\nabla^2 x$	<MathFullForm `lap[char[x]]'>
$\ln x$	<MathFullForm `ln[char[x]]'>
$(x$	<MathFullForm `lparen[char[x]]'>
$[x$	<MathFullForm `lparen[(*i1i*)char[x]]'>
$\{x$	<MathFullForm `lparen[(*i2i*)char[x]]'>
$-x$	<MathFullForm `minus[char[x]]'>
$\mp x$	<MathFullForm `mp[char[x]]'>
$\neg x$	<MathFullForm `neg[char[x]]'>
$\|x\|$	<MathFullForm `norm[char[x]]'>

Example	MathFullForm statement
$\bar{x}$	<MathFullForm `overline[char[x]]'>
∂x	<MathFullForm `partial[char[x]]'>
$\pm x$	<MathFullForm `pm[char[x]]'>
real$x$	<MathFullForm `real[char[x]]'>
$x)$	<MathFullForm `rparen[char[x]]'>
$x]$	<MathFullForm `rparen[(*i1i*)char[x]]'>
$x\}$	<MathFullForm `rparen[(*i2i*)char[x]]'>
sec$x$	<MathFullForm `sec[char[x]]'>
sech$x$	<MathFullForm `sech[char[x]]'>
;x	<MathFullForm `semicolon[char[x]]'>
sgn$x$	<MathFullForm `sgn[char[x]]'>
sin$x$	<MathFullForm `sin[char[x]]'>
sinh$x$	<MathFullForm `sinh[char[x]]'>
tan$x$	<MathFullForm `tan[char[x]]'>
tanh$x$	<MathFullForm `tanh[char[x]]'>
$\therefore x$	<MathFullForm `therefore[char[x]]'>
„x	<MathFullForm `ucomma[char[x]]'>
$= x$	<MathFullForm `uequal[char[x]]'>
$\underbrace{x}$	<MathFullForm `upbrace[char[x]]'>
δx	<MathFullForm `var[char[x]]'>

Binary operators

Binary operators have two operand expressions. One of the binary operators, `sn` (scientific notation), has two display formats. The following table contains an example of each binary operator with `char[x]` as a sample operand.

Example	MathFullForm statement
$\{x,x\}$	<MathFullForm `acmut[char[x],char[x]]'>
$x \bullet x$	<MathFullForm `bullet[char[x],char[x]]'>

Example	MathFullForm statement
$\langle x\|x\rangle$	`<MathFullForm `bket[char[x],char[x]]'>`
$\binom{x}{x}$	`<MathFullForm `choice[char[x],char[x]]'>`
$[x,x]$	`<MathFullForm `cmut[char[x],char[x]]'>`
$x \times x$	`<MathFullForm `cross[char[x],char[x]]'>`
$x \div x$	`<MathFullForm `div[char[x],char[x]]'>`
x/x	`<MathFullForm `fract[char[x],char[x]]'>`
$x(x)$	`<MathFullForm `function[char[x],char[x]]'>`
$\frac{\partial x}{\partial x}$	`<MathFullForm `function[oppartial[char[x]],char[x]]'>`[a]
$\frac{dx}{dx}$	`<MathFullForm `function[optotal[char[x]],char[x]]'>`
(x,x)	`<MathFullForm `inprod[char[x],char[x]]'>`
$\lim_{x} x$	`<MathFullForm `lim[char[x],char[x]]'>`
$\frac{x}{x}$	`<MathFullForm `over[char[x],char[x]]'>`
x^x	`<MathFullForm `power[char[x],char[x]]'>`
$x \times 10^x$	`<MathFullForm `sn[char[x],char[x]]'>`
$x\mathrm{E}x$	`<MathFullForm `sn[(*i1i*)char[x],char[x]]'>`

a. Partial and full differentials are a special case of `function`.

N-ary operators

N-ary operators have two or more operand expressions. When one of these operators has more than two operands, FrameMaker displays an additional operand symbol for each operand expression. For example, the following table shows several forms of `plus`.

Example	MathFullForm statement
$1 + 2$	`<MathFullForm `plus[num[1,"1"],num[2,"2"]]'>`
$1 + 2 + 3$	`<MathFullForm `plus[num[1,"1"],num[2,"2"],num[3,"3"]]'>`
$1 + 2 + 3 + 4$	`<MathFullForm` `` `plus[num[1,"1"],num[2,"2"],num[3,"3"],num[4,"4"]]'> ``

The following table contains an example of each n-ary operator. Each example shows two operands.

Example	MathFullForm statement
x x	`<MathFullForm `atop[char[x],char[x]]'>`
$x \approx x$	`<MathFullForm `approx[char[x],char[x]]'>`
$x \cap x$	`<MathFullForm `cap[char[x],char[x]]'>`
$x \cdot x$	`<MathFullForm `cdot[char[x],char[x]]'>`
x, x	`<MathFullForm `comma[char[x],char[x]]'>`
$x \cong x$	`<MathFullForm `cong[char[x],char[x]]'>`
$x \cup x$	`<MathFullForm `cup[char[x],char[x]]'>`
$x = x$	`<MathFullForm `equal[char[x],char[x]]'>`
$x \equiv x$	`<MathFullForm `equiv[char[x],char[x]]'>`
$x \geq x$	`<MathFullForm `geq[char[x],char[x]]'>`
$x \gg x$	`<MathFullForm `gg[char[x],char[x]]'>`
$x > x$	`<MathFullForm `greaterthan[char[x],char[x]]'>`
$x \in x$	`<MathFullForm `in[char[x],char[x]]'>`
$x \Diamond x$	`<MathFullForm `jotdot[char[x],char[x]]'>`
$x \leftarrow x$	`<MathFullForm `leftarrow[char[x],char[x]]'>`
$x \Leftarrow x$	`<MathFullForm `Leftarrow[char[x],char[x]]'>`
$x \leq x$	`<MathFullForm `leq[char[x],char[x]]'>`
$x < x$	`<MathFullForm `lessthan[char[x],char[x]]'>`
$x \quad x$	`<MathFullForm `list[char[x],char[x]]'>`
$x \ll x$	`<MathFullForm `ll[char[x],char[x]]'>`
$x \leftrightarrow x$	`<MathFullForm `lrarrow[char[x],char[x]]'>`
$x \Leftrightarrow x$	`<MathFullForm `LRarrow[char[x],char[x]]'>`
$x \ni x$	`<MathFullForm `ni[char[x],char[x]]'>`
$x \neq x$	`<MathFullForm `notequal[char[x],char[x]]'>`

Example	MathFullForm statement
$x \notin x$	<MathFullForm `notin[char[x],char[x]]'>
$x \not\subset x$	<MathFullForm `notsubset[char[x],char[x]]'>
$x \oplus x$	<MathFullForm `oplus[char[x],char[x]]'>
$x \otimes x$	<MathFullForm `otimes[char[x],char[x]]'>
$x \parallel x$	<MathFullForm `parallel[char[x],char[x]]'>
$x \perp x$	<MathFullForm `perp[char[x],char[x]]'>
$x + x$	<MathFullForm `plus[char[x],char[x]]'>
$x - x$	<MathFullForm `plus[char[x],minus[char[x]]]'>
$x \propto x$	<MathFullForm `propto[char[x],char[x]]'>
$x \rightarrow x$	<MathFullForm `rightarrow[char[x],char[x]]'>
$x \Rightarrow x$	<MathFullForm `Rightarrow[char[x],char[x]]'>
$x \sim x$	<MathFullForm `sim[char[x],char[x]]'>
$x \subset x$	<MathFullForm `subset[char[x],char[x]]'>
$x \subseteq x$	<MathFullForm `subseteq[char[x],char[x]]'>
$x \supset x$	<MathFullForm `supset[char[x],char[x]]'>
$x \supseteq x$	<MathFullForm `supseteq[char[x],char[x]]'>
xx	<MathFullForm `times[char[x],char[x]]'>
$x \vee x$	<MathFullForm `vee[char[x],char[x]]'>
$x \wedge x$	<MathFullForm `wedge[char[x],char[x]]'>

Large operators

Large operator expressions have one primary operand. In addition, they can have one or two range operands. The following table contains an example of each large operator with only one operand with `char[x]` as a sample operand.

Example	MathFullForm statement
$\bigcap x$	<MathFullForm `bigcap[char[x]]'>
$\bigcup x$	<MathFullForm `bigcup[char[x]]'>
$\int x$	<MathFullForm `int[char[x]]'>

Example	MathFullForm statement
$\oint x$	`<MathFullForm `oint[char[x]]'>`
$\prod x$	`<MathFullForm `prod[char[x]]'>`
$\sum x$	`<MathFullForm `sum[char[x]]'>`

Expressions with range operands have multiple display formats that change how operands are positioned around the symbol. Extended unions and intersections have two display formats. The formats are the same for both expressions; as an example, the following table shows the two display formats for an intersection with three operands:

Example	MathFullForm statement
$\bigcap_{2}^{3} 1$	`<MathFullForm` `` `bigcap[num[1.0,"1"],num[2.0,"2"],num[3.0,"3"]]'> ``
$\bigcap_{2}^{3} 1$	`<MathFullForm` `` `bigcap[(*i1i*)num[1.0,"1"],num[2.0,"2"],num[3.0,"3"]]'> ``

Sums, products, and integrals have three display formats. The formats are the same for all of these operators; as an example, the following table shows the display formats for an integral with three operands.

Example	MathFullForm statement
$\int_{a}^{b} x$	`<MathFullForm `int[char[x],char[a],char[b]]'>`
$\int_{a}^{b} x$	`<MathFullForm `int[(*i1i*)char[x],char[a],char[b]]'>`
$\int_{a}^{b} x$	`<MathFullForm `int[(*i2i*)char[x],char[a],char[b]]'>`

Expressions with optional operands

Some expressions have optional operands. In these expressions, the optional operands follow the primary operand. The following table contains an example of each expression with optional operands.

Example	MathFullForm statement
∇x	`<MathFullForm `grad[char[x]]'>`
$\nabla_2 1$	`<MathFullForm `grad[num[1,"1"],num[2,"2"]]'>`
$\log x$	`<MathFullForm `log[char[x]]'>`
$\log_x x$	`<MathFullForm `log[char[x],char[x]]'>`

Example	MathFullForm statement
$\frac{\partial}{\partial x}$	`<MathFullForm `oppartial[char[x]]'>`
$\frac{\partial^x}{\partial x}$	`<MathFullForm `oppartial[char[x],char[x]]'>`
$\frac{\mathrm{d}}{\mathrm{d}x}$	`<MathFullForm `optotal[char[x]]'>`
$\frac{\mathrm{d}^x}{\mathrm{d}x}$	`<MathFullForm `optotal[char[x],char[x]]'>`
$\sqrt{x}$	`<MathFullForm `sqrt[char[x]]'>`
${}^{x}\sqrt{x}$	`<MathFullForm `sqrt[char[x],char[x]]'>`
$x\vert$	`<MathFullForm `substitution[char[x]]'>`
$x\vert_x$	`<MathFullForm `substitution[char[x],char[x]]'>`
$x\vert_x^x$	`<MathFullForm `substitution[char[x],char[x],char[x]]'>`

For partial and full differentials (such as $\frac{\partial x}{\partial x}$ and $\frac{dx}{dx}$), see page 205.

Indexes

There are three expressions for describing indexes: `indexes`, `chem`, and `tensor`.

indexes: The `indexes` expression describes any number of subscripts and superscripts. The first operand is the number of superscripts and the second operand is the number of subscripts. Subsequent operands define the subscripts and then the superscripts.

Note: *Note that the number of superscripts is listed before the number of subscripts. However, superscript operands are listed after subscript operands.*

The following table contains an example of each `indexes` form.

Example	MathFullForm statement
x_1	`<MathFullForm `indexes[0,1,char[x],num[1,"1"]]'>`
x_{12}	`<MathFullForm `indexes[0,2,char[x],num[1,"1"],num[2,"2"]]'>`
x_1	`<MathFullForm `indexes[1,0,char[x],num[1,"1"]]'>`
x^{12}	`<MathFullForm `indexes[2,0,char[x],num[1,"1"],num[2,"2"]]'>`
x_1^2	`<MathFullForm `indexes[1,1,char[x],num[1,"1"],num[2,"2"]]'>`

Example	MathFullForm statement
x_{12}^{34}	<MathFullForm `indexes[2,2,char[x],num[1,"1"],num[2,"2"],num[3,"3"],num[4,"4"]]'>

chem: The `chem` expression defines pre-upper and pre-lower indexes, subscripts, and superscripts. Each position can have one expression. The following table shows all possible forms of `chem`.

Example	MathFullForm statement
${}_{1}^{}x$	<MathFullForm `chem[1,0,0,0,char[x],num[1,"1"]]'>
${}_{}^{1}x$	<MathFullForm `chem[0,0,1,0,char[x],num[1,"1"]]'>
${}_{2}^{1}x$	<MathFullForm `chem[1,0,1,0,char[x],num[1,"1"],num[2,"2"]]'>
${}_{1}^{}x_{2}$	<MathFullForm `chem[1,1,0,0,char[x],num[1,"1"],num[2,"2"]]'>
${}_{}^{1}x^{2}$	<MathFullForm `chem[0,0,1,1,char[x],num[1,"1"],num[2,"2"]]'>
${}_{2}^{1}x_{3}$	<MathFullForm `chem[1,1,1,0,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
${}_{3}^{1}x^{2}$	<MathFullForm `chem[1,0,1,1,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
${}_{3}^{1}x_{4}^{2}$	<MathFullForm `chem[1,1,1,1,char[x],num[1,"1"],num[2,"2"],num[3,"3"],num[4,"4"]]'>

tensor: The `tensor` expression represents specially formatted tensor notation. The first operand describes the position of the tensor indexes; subsequent operands define the indexes. The leftmost tensor index corresponds to the least significant bit of the first operand in binary format; the rightmost index corresponds to the most significant bit. 0 is the subscript position; 1 is the superscript position. The following table shows forms of `tensor`.

Example	MathFullForm statement
x_{1}^{2}	<MathFullForm `tensor[2,char[x],num[1,"1"],num[2,"2"]]'>
x^{1}_{2}	<MathFullForm `tensor[1,char[x],num[1,"1"],num[2,"2"]]'>
x^{1}_{23}	<MathFullForm `tensor[1,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
x_{1}^{23}	<MathFullForm `tensor[6,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
$x_{1\ 3}^{2}$	<MathFullForm `tensor[2,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
$x^{1\ 3}_{2}$	<MathFullForm `tensor[5,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>

Example	MathFullForm statement
x_{12}^{3}	<MathFullForm `tensor[4,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>
x_{3}^{12}	<MathFullForm `tensor[3,char[x],num[1,"1"],num[2,"2"],num[3,"3"]]'>

Matrices

The `matrix` expression defines a matrix. The first operand is the number of rows in the matrix; the second operand is the number of columns. Subsequent operands are expressions representing the elements of the matrix. The elements are listed from left to right and from top to bottom. The `matrix` expression has an alternate display format. The following table shows examples of `matrix`.

Example	MathFullForm statement
$\begin{bmatrix} x \end{bmatrix}$	<MathFullForm `matrix[1,1,char[x]]'>
x	<MathFullForm `matrix[(*i1i*)1,1,char[x]]'>
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$	<MathFullForm `matrix[2,3,num[1,"1"],num[2,"2"],num[3,"3"],num[4,"4"],num[5,"5"],num[6,"6"]]'>
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$	<MathFullForm `matrix[3,2,num[1,"1"],num[2,"2"],num[3,"3"],num[4,"4"],num[5,"5"],num[6,"6"]]'>

Custom operators

The following expressions allow you to use custom operators that have been defined on a math reference page:

Expression	Definition
`newinfix[x,y]`	Inserts custom infix operator
`newprefix[x]`	Inserts custom prefix operator
`newpostfix[x]`	Inserts custom postfix operator
`newfunction[x]`	Inserts custom function operator
`newlarge[x,y,z]`	Inserts custom large element
`newdelimiter[x]`	Inserts custom delimiter
`newlimit[x,y]`	Inserts custom limit function
`newvlist[x,y,z]`	Inserts custom vertical list

The expressions that insert new custom operators must include the name of the custom operator from the reference page. For example, suppose a document has a custom operator `MyFunction` that is added to the `DMathCatalog` statement as follows:

```
<DMathCatalog
    <DMathNew
```

```
                    # Names the new operator
        <DMathOpName `MyFunction'>
                    # Specifies the operator type
        <DMathNewType Function>
    >               # end of DMathNew
>                   # end of DMathCatalog
```

The corresponding `MathFullForm` statement appears as follows:

```
<MathFullForm `newfunction[(*T"MyFunction"T*)[char[x]]]'>
```

You do not use one of the custom operator expressions to insert a redefined math operator in an equation. Instead, you use the expression for the built-in operator, but force FrameMaker to use the new symbol from the reference page. For example, suppose you redefine the built-in operator `asin` and add it to the Math Catalog as follows:

```
<DMathCatalog
    <DMathOpOverrides
                    # Names the built-in operator
        <DMathOpName `asin'>
                    # Forces lookup from reference page
    <DMathOpTLineOverride Yes>
    >               # end of DMathOpOverrides
>                   # end of DMathCatalog
```

You would use the following `MathFullForm` statement:

```
<MathFullForm `asin[(*T"Inverse Sine"T*)operands]'>
```

where the string "`Inverse Sine`" is the name given to the frame on the reference page.

Sample equations

The following examples show `MathFullForm` statements for complete equations.

Example 1

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
<MathFullForm
`equal[char[x],over[plus[minus[char[b]],pm[sqrt[plus[power[char[b],num[2,"2"]],minus[times
[num[4,"4"],char[a],char[c]]]]]]],times[num[2,"2"],char[a]]]'>
```

Chapter 7: MIF Asian Text Processing Statements

This chapter describes the MIF statements used to express Asian text in a document. It includes character encoding statements, combined Asian and Western fonts, Kumihan tables, and rubi text.

Asian Character Encoding

Western text in a MIF file is written out as 7-bit ASCII. However, 7-bit encoding is insufficient for Asian text. Asian text in MIF files is represented by double-byte encoding. There are different encoding schemes for each supported language, and the MIF file must include a statement that can be used to determine which encoding to use.

The MIF file can be edited with an Asian-enabled text editor on the platform on which the MIF was written. If the text in a MIF file is in more than one Asian language, then only the language of the MIF encoding statement will be directly readable in a text editor. All other non 7-bit ASCII text will be backslashed escaped using the MIF backslash x convention.

MIFEncoding statement for Japanese

Adobe® FrameMaker® recognizes two encoding schemes for Japanese; Shift-JIS and EUC. The Windows versions of FrameMaker write Shift-JIS for Japanese text, and the UNIX versions of FrameMaker write out EUC. The MIF can converted between Shift-JIS and EUC using a Japanese text conversion utility. The MIF encoding statement is converted along with the text in the MIF file.

To determine which encoding was used, each MIF file that contains Japanese text must include a `MIFEncoding` statement near the beginning of the file. It must appear before any Japanese text in the file. The string value in the `MIFEncoding` statement is the Japanese spelling of the word "Nihongo," which means Japanese. FrameMaker reads this fixed string and determines what the encoding is for it. From that, FrameMaker expects the same encoding to be used for all subsequent 8-bit text in the document.

To see the characters spelling the word Nihongo, you must view the MIF file on a system that is enabled for Japanese character display. When the MIF is displayed on a Roman system, the characters appear garbled.

Syntax

```
<MIFEncoding `日本語'> # originally written as Japanese (Shift-JIS)
<MIFEncoding `日本語'> # originally written as Japanese (EUC)
```

MIFEncoding statement for Chinese

FrameMaker recognizes three encoding schemes for Chinese; Big5 and CNS for Traditional Chinese, and GB2312-80 for Simplified Chinese. The Windows versions of FrameMaker write Big5 for Traditional Chinese text, and the UNIX versions of FrameMaker write out CNS for Traditional Chinese text. All platform versions of FrameMaker write GB2312-80 for Simplified Chinese.

To determine which encoding was used, each MIF file that contains Chinese text must include a `MIFEncoding` statement near the beginning of the file. It must appear before any Chinese text in the file. The string value in the `MIFEncoding` statement is the Chinese spelling of the word "Chinese". FrameMaker reads this fixed string and determines what the hexadecimal encoding is for it. From that, FrameMaker expects the same encoding to be used for all subsequent Asian text in the document.

To see the characters spelling the word "Chinese", you must view the MIF file on a system that is enabled for Chinese character display. When the MIF is displayed on a Roman system, the characters appear garbled.

Syntax

```
<MIFEncoding `中文 '> # originally written as Traditional Chinese (Big5)
<MIFEncoding `中文 '> # originally written as Traditional Chinese (CNS)
<MIFEncoding `中文 '> # originally written as Simplified Chinese
```

MIFEncoding statement for Korean

FrameMaker recognizes one encoding scheme for Korean: KSC5601. All platform versions of FrameMaker write KSC5601 for Korean.

Each MIF file that contains Korean text must include a `MIFEncoding` statement near the beginning of the file. It must appear before any Korean text in the file. The string value in the `MIFEncoding` statement is the Korean spelling of the word "Korean." FrameMaker reads this fixed string and determines what the hexadecimal encoding is for it. From that, FrameMaker expects the same encoding to be used for all subsequent Asian text in the document.

To see the characters spelling the word "Korean.", you must view the MIF file on a system that is enabled for Korean character display. When the MIF is displayed on a Roman system, the characters appear garbled.

Syntax

```
<MIFEncoding `한국어'> # originally written as Korean
```

Combined Fonts

Combined fonts assign two component fonts to one combined font name. This is done to handle both an Asian font and a Western font as though they are in one font family. In a combined font, the Asian font is the base font, and the Roman font is the Western font. For example, you can create a combined font named Mincho-Palatino that uses Mincho for Asian characters and switches to Palatino for Roman characters.

When reading a MIF paragraph that uses Mincho-Palatino, FrameMaker displays Asian characters in Mincho and Roman characters in Palatino. If the Mincho font is not installed on the user's system, FrameMaker displays the Asian text in a font that uses the same character encoding as Mincho.

CombinedFontCatalog statement

Combined fonts are defined for the document in the `CombinedFontCatalog` statement. For each combined font, there is a `CombinedFontDefn` statement that specifies the combined font name and identifies the Asian and the Roman component fonts. Note that the combined font catalog must precede the first `PgfFont` and `Font` statements in the document.

Syntax

```
<CombinedFontCatalog
```

<CombinedFontDefn	Defines a single combined font
<CombinedFontName *string*>	The name of the combined font
<CombinedFontBaseFamily *string*>	The name of the Asian component font
<CombinedFontWesternFamily *string*>	The name of the Roman component font
<CombinedFontWesternSize *percent*>	The size of the Roman component font, expressed as a percentage of the base font size; allowed values are 1.0% through 1000.0%
<CombinedFontWesternShift *percent*>	The baseline offset of the Roman font, expressed as a percentage of the base font size where a positive value raises the Roman baseline above the Asian baseline; allowed values are -1000.0% through 1000.0%
<CombinedFontBaseEncoding *keyword*>	Specifies the encoding for the base font. *keyword* can be one of: JISX0208.ShiftJIS BIG5 GB2312-80.EUC KSC5601-1992
<CombinedFontAllowBaseFamilyBoldedAndObliqued *boolean*>	Yes allows a simulation of the bold or italic Asian component font to be used if Bold or Italic/Oblique is applied to the combined font.
>	End of the CombinedFontDefn statement
...	More CombinedFontDefn statements as needed
>	End of the CombinedFontCatalog statement

Example

The following is an example of a combined font catalog:

```
<CombinedFontCatalog
 <CombinedFontDefn
  <CombinedFontName `MyCombinedFont'>
  <CombinedFontBaseFamily `Osaka'>
  <CombinedFontWesternFamily `Times'>
  <CombinedFontWesternSize  75.0%>
  <CombinedFontWesternShift  0.0%>
  <CombinedFontBaseEncoding `JISX0208.ShiftJIS'>
  <CombinedFontAllowBaseFamilyBoldedAndObliqued Yes>
 >              # end of CombinedFontDefn
>               # end of CombinedFontCatalog
```

PgfFont or Font statement

When a combined font is used in a paragraph or text line, the PgfFont or Font statement includes the combined font name and the base font's family name. These statements also include the PostScriptName and PlatformName for both the base and the Roman fonts.

FCombinedFontName is a new statement to express the combined font name. The FFamily statement expresses the base font's family name.

The FPostScriptName and FPlatformName statements all refer to the base font. The following new statements have been added to express the corresponding values for the Roman font:

- FWesternPostScriptName
- FWesternPlatformName

Syntax

<PgfFont	
…	
<FPostScriptName *string*>	The PostScript name for the base font
<FPlatformName *string*>	The platform name for the base font
<FWesternPostScriptName *string*>	The PostScript name for the Roman font
<FWesternPlatformName *string*>	The platform name for the Roman font
<FCombinedFontName *string*>	The name of the combined font, as defined in the combined font catalog
<FEncoding *string*>	Specifies the encoding for the base font. This is to specify the encoding for a double-byte font. If not present, the default is Roman. *keyword* can be one of: JISX0208.ShiftJIS BIG5 GB2312-80.EUC KSC5601-1992
…	
>	End of the PgfFont statement

Example

The following is an example of a combined font in a Para statement:

```
<Para
  <Unique 996885>
  <PgfTag `Body'>
  <ParaLine
   <Font
    <FTag `'>
    <FPlatformName `M.Osaka.P'>
    <FWesternPlatformName `M.Times.P'>
    <FFamily `Osaka'>
    <FCombinedFontName `MyCombinedFont'>
    <FEncoding `JISX0208.ShiftJIS'>
    <FLocked No>
   >               # end of Font
   <String `CombinedFontStatement '>
   <Font
    <FTag `'>
    <FPlatformName `M.Osaka.P'>
    <FWesternPlatformName `M.Times.P'>
    <FFamily `Osaka'>
    <FCombinedFontName `MyCombinedFont'>
```

```
    <FWeight `Medium'>
    <FEncoding `JISX0208.ShiftJIS'>
    <FLanguage Japanese>
    <FLocked No>
   >                # end of Font
   <String `組版'>
  >                 # end of ParaLine
 >                  # end of Para
```

Kumihan Tables

Kumihan tables specify line composition rules for Japanese documents. FrameMaker uses standard JIS 4051 Kumihan rules by default. In most cases, the JIS standard is fine, but there are cases where corporate standards might differ from the JIS rules.

Kumihan tables are associated with a document. To customize the Kumihan tables for a document, you specify the tables in MIF. Then you can import the MIF into an existing document, or into a template you will use to create new documents.

Understanding Kumihan tables

Kumihan tables specify line composition rules by assigning characters to various classes, and then specifying four tables of rules that apply to the characters of each class.

The `CharClass` statement assigns each character to one of 25 classes. For example, the `BegParentheses class` and the `EndParentheses` class are defined by the following MIF statements, and they contain the characters shown in the statement.

```
<BegParentheses `‘“〈《「『【〔［｛〈《「『【〔［｛
<EndParentheses `’”〉》」』】〕］｝〉》」』】〕］｝
```

For more information on the `CharClass` statement, see “CharClass statement” on page 219.

The four statements that define the tables of rules that apply to the characters of each class are `SqueezeTable`, `SpreadTable`, `LineBreakTable`, and `ExtraSpaceTable`. Each of these statements specify the actions FrameMaker takes for the characters in each of the 25 classes.

For example, the `LineBreakTable` statement specifies whether a line break can occur between a character of one class and a character of another class. Here is an example of a `LineBreakTable` statement that specifies when a line break can occur between a character in the `BegParentheses` class and a character in each of the 25 classes:

```
<BegParentheses  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1>
```

The 25 numerical values for the `BegParentheses` statement specify the actions FrameMaker takes when a character from each of the 25 classes, such as an ending parenthesis character, follows a character in the `BegParentheses` class. The position of each numerical value after the `BegParentheses` statement specifies the class. For example, the first position is the `BegParentheses` class, the `EndParentheses` class is the second position, and so on. If a numerical value of 0 is specified, FrameMaker allows a line break between a character the `BegParentheses` class and a character in the class specified in that position in the statement. If a value of 1 is specified, FrameMaker does not allow a line break.

```
                    BegParentheses  EndParentheses  NoLineBeginChar  QuestionBang  CenteredPunct  PeriodComma  NonSeparableChar  PrecedingSymbol  SucceedingSymbol  AsianSpace  Hiragana  Others  BaseCharWithSuper  BaseCharWithRubi  Numeral  UnitSymbol  RomanSpace  RomanChar  ParenBeginWariChu  ParenEndWariChu  Spare1  Spare2  Spare3  Spare4  Spare5
<BegParentheses      1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3  1  1  1  1  1
```

The column position of each numerical value in the statement specifies the action to take for each class.

In the preceding example, a line break does not occur between a character in the `BegParentheses` class and a character in the `EndParentheses` class because the value 1 is in the second position, which is the column position for the EndParentheses class of characters. For more information on the `LineBreakTable` statement, see "LineBreakTable statement" on page 224.

Writing Kumihan tables as MIF

FrameMaker only writes out Kumihan tables in MIF when you are running FrameMaker on Asian system software. If you are running on an Asian system, when you save a document as MIF, the Kumihan tables are written out as part of the document.

This is most critical with the character classes. To specify a character class in MIF, you must be able to type the character and save it in a text file. The standard Western system doesn't include these character sets in its character code page, so these characters would appear garbled. You need the Asian system to represent the characters in a text file.

To see an example of a Kumihan table, it is best to save a document as MIF, open the MIF on an Asian system in a text editor, and search for the `KumihanCatalog` statement.

Specifying Kumihan tables in MIF

The following statements specify the Kumihan catalog and all of its component tables.

KumihanCatalog statement

The `KumihanCatalog` statement begins the Kumihan table specification for the document. Note that the Kumihan catalog is not included in the `<Document>` block, but is in a block of its own.

Each Asian language can have its own Kumihan tables. This means that one Kumihan catalog can have up to four sets of tables, one set for each of the four supported Asian languages (Japanese, Traditional Chinese, SimpleChinese, and Korean).

Syntax

`<KumihanCatalog`	
`<Kumihan`	Defines a Kumihan table set
`...`	
`<Kumihan`	Additional Kumihan table sets as needed (one for each Asian language - up to four per document)
`...`	
`>`	End of `KumihanCatalog` statement

Kumihan statement

The `Kumihan` statement defines a set of Kumihan tables. A document can have one set of tables for each of the four supported Asian languages.

Syntax

`<Kumihan`	Defines a Kumihan table

`<Klanguage` *keyword*`>`	The language for this table
	keyword can be one of:
	`Japanese`
	`TraditionalChinese`
	`SimpleChinese`
	`Korean`

`<CharClass`	Defines character class assignments
`...`	
`<SqueezeTable`	Defines the squeeze table
`...`	
`<SpreadTable`	Defines the spread table
`...`	
`<LineBreakTable`	Defines the line break table
`...`	
`<ExtraSpaceTable`	Defines the extra space table

CharClass statement

The `CharClass` statement assigns individual characters to one of 25 classes. The JIS standard recognizes 20 classes, and MIF includes an additional five classes (`Spare1` through `Spare5`) so you can assign characters custom character classes.

MIF Statement	Column Position	Description
`<CharClass`		

MIF Statement	Column Position	Description
<BegParentheses *chars*>	1	The characters to use as opening parentheses
<EndParentheses *chars*>	2	The characters to use as ending parentheses
<NoLineBeginChar *chars*>	3	Characters that cannot start a new line of text
<QuestionBang *chars*>	4	Characters for questions and exclamations
<CenteredPunct *chars*>	5	Punctuation characters that must be centered between characters
<PeriodComma *chars*>	6	Punctuation that is not centered
<NonSeparableChar *chars*>	7	Characters that cannot have line breaks between them
<PrecedingSymbol *chars*>	8	Characters such as currency symbols (¥ or $)
<SucceedingSymbol *chars*>	9	Characters such as % or ° (degree)
<AsianSpace *chars*>	10	Characters for spaces in Asian text
<Hiragana *chars*>	11	The set of hiragana characters
<Others>	12	All characters not assigned to any class automatically belong to <Others>
<BaseCharWithSuper *chars*>	13	FrameMaker uses this class to allow spreading between the end of a footnote and the next character. Do not assign any characters to this class.
<BaseCharWithRubi *chars*>	14	The rubi block, including oyamoji and rubi text. This class has to do with Rubikake and Nibukake rules that specify how to handle spacing between a rubi block and an adjacent character.
<Numeral *chars*>	15	Characters for numerals
<UnitSymbol *chars*>	16	This class is not used by FrameMaker
<RomanSpace *chars*>	17	Characters for spaces in Roman text
<RomanChar *chars*>	18	Characters for Roman text
<ParenBeginWariChu *chars*>	19	The current version of FrameMaker does not support Warichu; this class is not used by FrameMaker
<ParenEndWariChu *chars*>	20	The current version of FrameMaker does not support Warichu; this class is not used by FrameMaker
<Spare1 *chars*>	21	Reserved for a user-defined character class
<Spare2 *chars*>	22	Reserved for a user-defined character class
<Spare3 *chars*>	23	Reserved for a user-defined character class
<Spare4 *chars*>	24	Reserved for a user-defined character class
<Spare5 *chars*>	25	Reserved for a user-defined character class
>		End of the CharClass statement

Usage

Assigning characters to a class identifies them in the succeeding tables so the various typographical rules can be specified for each class of character.

Any character that is not assigned to a class is automatically assigned to the `<Others>` class. When specifying classes, you should not assign any characters to `<Others>`. In fact, it is not necessary to include a MIF statement for `<Others>`. In the following tables, the 12th column position corresponds to the `<Others>` class.

If you are using Asian system software, you can enter the characters for each class directly in a text file.

Example

The following is an example of a portion of a `CharClass` statement:

```
<CharClass
 <BegParentheses `‘“〈《「『【〔［｛〈《「『【〔［｛｢'>
 <EndParentheses `’”〉》」』】〕］｝〉》」』】〕］｝｣'>
 <NoLineBeginChar `ヽヾゝゞ々ーぁぃぅぇぉっゃゅょゎァィゥェォッャュョヮカケ'>
 ...
>                  # end of CharClass
```

SqueezeTable statement

The `SqueezeTable` statement defines how to compress the space surrounding characters of each class. Note that each character is rendered within a specific area. For Asian characters, this area is the same for each character. These rules determine how to compress this area for optimum line rendering.

Syntax

`<SqueezeTable`	
`<SqueezeHorizontal` *numerals*`>`	Defines how to squeeze horizontal text
`<SqueezeVertical` *numerals*`>`	Defines how to squeeze vertical text
`>`	End of `SqueezeTable` statement
The possible values for *numerals* are: 0 - No squeeze 1 - Half squeeze from top or left 2 - Half squeeze from bottom or right 3 - Quarter squeeze from all sides 4 - Same as 3, but do not apply vertical squeeze to a semicolon 5 - This character pair should not have occurred	

Usage

The `SqueezeHorizontal` and `SqueezeVertical` statements include 25 numerical values, one for each character class. The values are separated by a space. An example of a squeeze table statement is:

```
<SqueezeTable
                          BegParentheses
                          EndParentheses
                          NoLineBeginChar
                          QuestionBang
                          CenteredPunct
                          PeriodComma
                          NonSeparableChar
                          PrecedingSymbol
                          SucceedingSymbol
                          AsianSpace
                          Hiragana
                          Others
                          BaseCharWithSuper
                          BaseCharWithRubi
                          Numeral
                          UnitSymbol
                          RomanSpace
                          RomanChar
                          ParenBeginWariChu
                          ParenEndWariChu
                          Spare1
                          Spare2
                          Spare3
                          Spare4
                          Spare5
 <SqueezeHorizontal       1 1 2 0 0 3 2 0 0 0 0 0 0 0 0 5 0 0 0 1 2 0 0 0 0 0
 <SqueezeVertical         1 2 0 0 4 2 0 0 0 0 0 0 0 0 5 0 0 0 1 2 0 0 0 0 0 0
> # end of SqueezeTable
```

In the preceding example, the `SqueezeHorizontal` value for a character in the `NoLineBeginChar` class is `2`, which specifies half squeeze from the right.

SpreadTable statement

The `SpreadTable` statement defines how to reduce the squeeze that was applied to adjacent characters. There are 25 statement rows in this table, each corresponding to the 25 character classes, respectively.

There are 26 numeric values in each statement row. The first 25 values correspond to the 25 character classes, respectively. The 26th value corresponds to the beginning or end of a line. These values specify how to spread a character of the class identified by the row statement, when followed by a character in the class identified by the column position in the statement.

Syntax

`<SpreadTable`
`<BegParentheses` *numerals*`>`
`<EndParentheses` *numerals*`>`
`<NoLineBeginChar` *numerals*`>`
`<QuestionBang` *numerals*`>`
`<CenteredPunct` *numerals*`>`
`<PeriodComma` *numerals*`>`
`<NonSeparableChar` *numerals*`>`
`<PrecedingSymbol` *numerals*`>`
`<SucceedingSymbol` *numerals*`>`
`<AsianSpace` *numerals*`>`
`<Hiragana` *numerals*`>`
`<Others>`

<BaseCharWithSuper *numerals*>
<BaseCharWithRubi *numerals*>
<Numeral *numerals*>
<UnitSymbol *numerals*>
<RomanSpace *numerals*>
<RomanChar *numerals*>
<ParenBeginWariChu *numerals*>
<ParenEndWariChu *numerals*>
<Spare1 *numerals*>
<Spare2 *numerals*>
<Spare3 *numerals*>
<Spare4 *numerals*>
<Spare5 *numerals*>
> End of `SpreadTable` statement

The possible values for *numerals* are:

0 - No spread
1 - Spread the first character of the pair by 1/2 em
2 - Spread the second character of the pair by 1/2 em
3 - Spread the first character of the pair by 1/4 em
4 - Spread the second character of the pair by 1/4 em
5 - Spread both characters of the pair by 1/4 em
6 - Spread the first character by 1/2 em and the second character by 1/4 em
7 - Add spread to the first character of an Asian/Roman character pair
8 - Add spread to the second character of a Roman/Asian character pair
9 - Delete the first occurance of the two spaces; for example, delete the first of two adjacent Roman space characters
10 - Nibukake - Rubi may extend over the preceding nibukake, but it cannot exceed the nibukake; add space to the first oyamoji character
11 - Nibukake - Rubi may extend over the following nibukake, but it cannot exceed the nibukake; add space to the last oyamoji character
12 - Allow rubi text to extend over oyamoji character when betagumi; no space is added
13 - Place oyamoji character with rubi based on the standard rule
14 - Double yakumono - Double yakumono rule is applied
15 - This character pair should not have occurred

Usage

Each statement row in the spread table includes 26 numerical values, one for each character class, and an added value for the characters at the beginning or the end of a line. The values are separated by a space. An example of a spread table is:

```
<SpreadTable
                        BegParentheses EndParentheses NoLineBeginChar QuestionBang CenteredPunct PeriodComma NonSeparableChar PrecedingSymbol SucceedingSymbol AsianSpace Hiragana Others BaseCharWithSuper BaseCharWithRubi Numeral UnitSymbol RomanSpace RomanChar ParenBeginWariChu ParenEndWariChu Spare1 Spare2 Spare3 Spare4 Spare5
 <BegParentheses        1 0 0 0 4 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1
                        4                                                2                                5                 5
 <EndParentheses        1 1 1 1 4 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
                          4         4                                0                                                 5
> # end of SpreadTable
```

In the preceding example, no spread occurs between a character in the `BegParentheses` class and a character in the `QuestionBang` class because the value `0` (No spread) is in the fourth position, which is the column position for the `QuestionBang` class of characters.

LineBreakTable statement

The `LineBreakTable` statement defines how to break lines between characters. There are 25 statement rows in this table, each corresponding to the 25 character classes, respectively.

There are 25 numeric values in each statement row. Each value corresponds to one of the 25 character classes, respectively. These values specify how to break a line after a character of the class identified by the row statement, when followed by a character of the class identified by the column position.

Syntax

<LineBreakTable
<BegParentheses *numerals*>
<EndParentheses *numerals*>
<NoLineBeginChar *numerals*>
<QuestionBang *numerals*>
<CenteredPunct *numerals*>
<PeriodComma *numerals*>
<NonSeparableChar *numerals*>
<PrecedingSymbol *numerals*>
<SucceedingSymbol *numerals*>
<AsianSpace *numerals*>

<Hiragana *numerals*>
<Others>
<BaseCharWithSuper *numerals*>
<BaseCharWithRubi *numerals*>
<Numeral *numerals*>
<UnitSymbol *numerals*>
<RomanSpace *numerals*>
<RomanChar *numerals*>
<ParenBeginWariChu *numerals*>
<ParenEndWariChu *numerals*>
<Spare1 *numerals*>
<Spare2 *numerals*>
<Spare3 *numerals*>
<Spare4 *numerals*>
<Spare5 *numerals*>
> End of `LineBreakTable` statement
The possible values for *numerals* are: 0 - Line break is allowed 1 - Line break is not allowed 2 - Break the line according to Roman text rules 3 - This character pair should not have occurred

Usage

Each statement row in the line break table includes 25 numerical values, one for each character class. The values are separated by a space. An example of a line break table is:

```
<LineBreakTable
                        BegParentheses
                        EndParentheses
                        NoLineBeginChar
                        QuestionBang
                        CenteredPunct
                        PeriodComma
                        NonSeparableChar
                        PrecedingSymbol
                        SucceedingSymbol
                        AsianSpace
                        Hiragana
                        Others
                        BaseCharWithSuper
                        BaseCharWithRubi
                        Numeral
                        UnitSymbol
                        RomanSpace
                        RomanChar
                        ParenBeginWariChu
                        ParenEndWariChu
                        Spare1
                        Spare2
                        Spare3
                        Spare4
                        Spare5
 <BegParentheses        1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3  0  0  0  0  0
 <EndParentheses        0  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0
 ...
> # end of LineBreakTable
```

In the preceding example, a line break can occur between a character in the `EndParentheses` class and a character in the `NonSeparableChar` class because the value `0` (Line break is allowed) is in the seventh position, which is the column position for the `NonSeparableChar` class of characters.

ExtraSpaceTable statement

The `ExtraSpaceTable` statement defines how to add extra space between characters when needed for full justification. There are 25 statement rows in this table, each corresponding to the 25 character classes, respectively.

There are 25 numeric values in each statement row. Each value corresponds to one of the 25 character classes, respectively. These values specify how to add space after a character of the class identified by the row statement, when followed by a character of the class identified by the column position.

Syntax

<ExtraSpaceTable
<BegParentheses *numerals*>
<EndParentheses *numerals*>
<NoLineBeginChar *numerals*>
<QuestionBang *numerals*>
<CenteredPunct *numerals*>
<PeriodComma *numerals*>
<NonSeparableChar *numerals*>
<PrecedingSymbol *numerals*>
<SucceedingSymbol *numerals*>
<AsianSpace *numerals*>
<Hiragana *numerals*>
<Others>
<BaseCharWithSuper *numerals*>
<BaseCharWithRubi *numerals*>
<Numeral *numerals*>
<UnitSymbol *numerals*>
<RomanSpace *numerals*>
<RomanChar *numerals*>
<ParenBeginWariChu *numerals*>
<ParenEndWariChu *numerals*>
<Spare1 *numerals*>
<Spare2 *numerals*>

<Spare3 *numerals*>	
<Spare4 *numerals*>	
<Spare5 *numerals*>	
>	End of `ExtraSpaceTable` statement

The possible values for *numerals* are:

0 - Extra space is allowed
1 - Extra space is not allowed
2 - Add extra space to the last character of a Roman word
3 - Add extra space after a Roman character
4 - Add extra space if the adjacent characters are one each of Japanese and Roman characters
5 - Delete one of two space characters. Note that FrameMaker does not use this action because the Smart Spaces feature performs it automatically
6 - This character pair should not have occurred

Usage

Each statement row in the extra space table includes 25 numerical values, one for each character class. The values are separated by a space. An example of a extra space table is:

```
<ExtraSpaceTable
                        BegParentheses
                        EndParentheses
                        NoLineBeginChar
                        QuestionBang
                        CenteredPunct
                        PeriodComma
                        NonSeparableChar
                        PrecedingSymbol
                        SucceedingSymbol
                        AsianSpace
                        Hiragana
                        Others
                        BaseCharWithSuper
                        BaseCharWithRubi
                        Numeral
                        UnitSymbol
                        RomanSpace
                        RomanChar
                        ParenBeginWariChu
                        ParenEndWariChu
                        Spare1
                        Spare2
                        Spare3
                        Spare4
                        Spare5
 <BegParentheses        1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  1  1  6  1  1  1  1  1
 <EndParentheses        1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  1  1  1  1  1  1  1  1
 ...
> # end of ExtraSpaceTable
```

In the preceding example, a extra space is not allowed between a character in the `EndParentheses` class and a character in the `CenteredPunct` class because the value `1` (Extra space is not allowed) is in the fifth position, which is the column position for the `CenteredPunct` class of characters.

Rubi text

Rubi text is a Japanese system for representing the pronunciation of words as a string of phonetic characters (hiragana) directly above the word in question (oyamoji). A MIF file includes document-level statements that describe the settings made in the Rubi Properties dialog box, as well as MIF statements for a rubi composite.

A rubi composite includes both oyamoji text and rubi text. If the document is structured, the rubi composite contains an object tagged RubiGroup, the oyamoji text, an element tagged Rubi, and the rubi text.

Document statement

In addition to document preferences (see "Document statement" on page 88), the MIF `Document` statement describes standard formats for rubi text. The rubi formatting substatements correspond to settings in the Rubi Properties dialog box.

Syntax

`<Document`	See page 88
`<DRubiSize` *`percentage`*`>` OR	The size of the rubi characters, proportional to the size of the oyamoji characters Allowed values are 1.0% through 1000.0%
`<DRubiFixedSize` *`point size`*	The fixed size of the rubi characters in points only. Either the DRubiSize statement or the DRubiFixedSize statement can be specified, but not both in the same document.
`<DRubiOverhang` *`boolean`*`>`	Yes allows rubi to overhang hiragana oyamoji text
`<DRubiAlignAtBounds` *`boolean`*`>`	Yes aligns all rubi and oyamoji characters at line boundaries

`<DWideRubiSpaceForJapanese` *`keyword`*`>`	Determines how to space rubi characters for Japanese oyamoji that is wider than the rubi text
	`keyword` can be:
	`Wide`
	`Narrow`
	`Proportional`

`<DNarrowRubiSpaceForJapanese` *`keyword`*`>`	Determines how to space rubi characters for Japanese oyamoji that is narrower than the rubi text
	`keyword` can be:
	`Wide`
	`Narrow`
	`Proportional`

`<DWideRubiSpaceForOther` *`keyword`*`>`	Determines how to space rubi characters for non-Japanese oyamoji that is wider than the rubi text
	`keyword` can be:
	`Wide`
	`Narrow`
	`Proportional`

`<DNarrowRubiSpaceForOther` *`keyword`*`>`	Determines how to space rubi characters for non-Japanese oyamoji that is narrower than the rubi text
	`keyword` can be:
	`Wide`
	`Narrow`
	`Proportional`

`>`	End of the `Document` statement

Example

```
<Document
    . . .
        <DRubiSize 50%>
        <DRubiOverhang Yes>
        <DRubiAlignAtBounds Yes>
        <DWideSpaceForJapanese Proportional>
```

```
        <DNarrowSpaceForJapanese Proportional>
        <DWideSpaceForOther Narrow>
        <DNarrowSpaceForOther Narrow>
    . . .
>                   # end of Document
```

RubiCompositeBegin statement

The `RubiCompositeBegin` statement is always matched with a `RubiCompositeEnd` statement. Between them are the contents of the rubi composite; the oyamoji and the rubi text. A rubi composite can occur anywhere in a `Paraline` statement. Also, anything that can occur within a `Paraline`, except another rubi composite, can also occur between the `RubiCompositeBegin` and `RubiCompositeEnd` statements.

In a structured document, the rubi composite includes a RubiGroup element and a Rubi element.

Syntax

`<RubiCompositeBegin>`	Starts the rubi composite
`<Element`	For structured documents only - Defines the RubiGroup element
`...`	Continue the RubiGroup element specification
`>`	End of the RubiGroup element
`<String` *`string`*`>`	The oyamoji text
`<RubiTextBegin>`	Begins the rubi text
`<Element`	For structured documents only - Defines the Rubi element
`...`	Continue the Rubi element specification
`>`	End of the Rubi element
`<String` *`string`*`>`	The rubi text
`<RubiTextEnd>`	Ends the rubi text
`<RubiCompositeEnd>`	Ends the rubi composite

Example - unstructured

```
<Paraline
<String ` kumihan '>
    . . .
<RubiCompositeBegin
        <String `組版'>
        <RubiTextBegin
        <String `おや文字'>
   <RubiTextEnd >
   <RubiCompositeEnd >
  >                 # end of ParaLine
```

Example - structured

```
<Paraline
    <String ‘Some text ’>
    . . .
        <RubiCompositeBegin
            <Element
                <Unique 123456>
```

```
            <ETag `RubiGroup'>
            <Attributes
                .       #. . Typical MIF to define attributes
            >           # end of Attributes
            <Collapsed No>
            <SpecialCase No>
            <AttributeDisplay AllAttributes>
        >               # end of Element
    >                   # end of RubiCompositeBegin
        <String `Oyamoji text'>
    <RubiTextBegin
        <Element
            <Unique 123457>
            <ETag `Rubi'>
            <Attributes
                .       #. . Typical MIF to define attributes
            >           # end of Attributes
            <Collapsed No>
            <SpecialCase No>
            <AttributeDisplay AllAttributes>
        >               # end of Element
        <String `Rubi text'>
    <RubiTextEnd>
<RubiCompositeEnd>
<String `Some more text '>
. . .
>                   # end of Paraline
```

Chapter 8: Examples

The examples in this appendix show how to describe text and graphics in MIF files. (The current examples are valid only for unstructured documents.) You can import the MIF file into an existing Adobe® FrameMaker® template, or you can open the MIF file as a FrameMaker document. In either case, if you save the resulting document in MIF format, you will create a complete description of the document—not just the text or graphics.

If you find any MIF statement difficult to understand, the best way to learn more is to create a sample file that uses the statement. Use FrameMaker to edit and format a document that uses the MIF feature and then save the document as a MIF file. Examine the MIF file with any standard text editor.

The examples in this appendix are provided online.

For FrameMaker on this platform	Look here
UNIX	`$FMHOME/fminit/`*`language`*`/Samples`, where *`language`* is the language in use, such as `usenglish`
Windows	The `samples` directory where *MIF Reference* is installed

Text example

This example shows a simple text file and the MIF file that describes it. If you are writing a filter program to convert text files to MIF, your program should create a similar MIF file. The following text file was created with a text editor:

```
MIF (Maker Interchange Format) is a group of statements that describe all text and
graphics understood by FrameMaker in an easily parsed, readable text file. MIF
provides a way to exchange information between FrameMaker and other applications
while preserving graphics, document structure, and format.
You can write programs that convert graphics or documents into a MIF file and then
import the MIF file into a FrameMaker document with the graphics and document
formats intact.
```

A filter program translated the text file to produce the following MIF file:

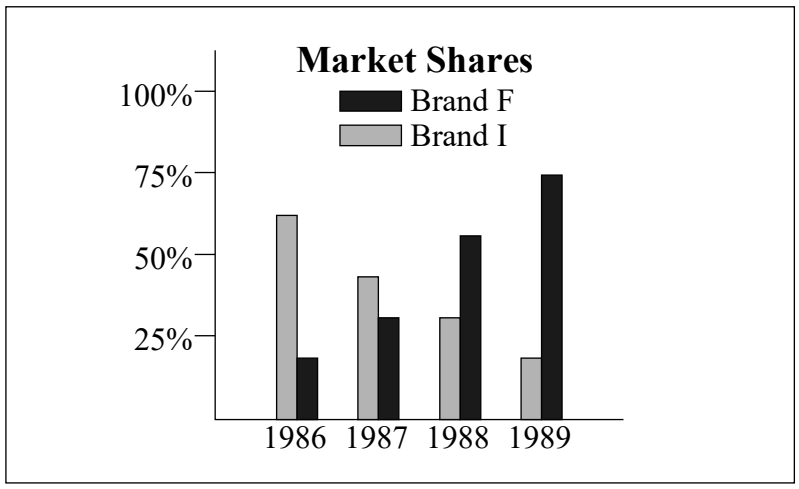
```
<MIFFile 2015>                      # Identifies this as a MIF file.
                                    # The macros below are used only for the second paragraph
of
                                    # text, to illustrate how they can ease the process of
                                    # MIF generation.
define(pr,`<Para ')
define(ep,`>')
define(ln,`<ParaLine <String')
define(en,`>>')
                                    # First paragraph of text.
<Para
                                    #
                                    # <PgfTag> statement forces a lookup in the document's
                                    # Paragraph Catalog, so you don't have to specify the
format
                                    # in detail here.
    <PgfTag `Body'>
```

```
                                    #
                                    # One <ParaLine> statement for each line in the
paragraph.
                                    # Line breaks don't matter; the MIF interpreter adjusts
line
                                    # breaks when the file is opened or imported.
    <ParaLine
        <String `MIF (Maker Interchange format) is a group of '>
    >
    <ParaLine
        <String ` statements that describe all text and graphics '>
    >
    <ParaLine
        <String `understood by FrameMaker in an easily parsed, '>
    >
    <ParaLine
        <String `readable text file. MIF provides a way to exchange '>
    >
    <ParaLine
        <String `information between FrameMaker and other ' >
    >
    <ParaLine
        <String `applications while preserving graphics, document '>
    >
    <ParaLine
        <String `structure, and format. ' >
    >
>                                   # end of Para
                                    #
                                    # Second paragraph of text.Macros defined earlier are
used
                                    # here.
                                    # This paragraph inherits the format of the previous one,
                                    # since there's no PgfTag or Pgf statement to override
it.
pr
ln `You can write programs that convert graphics or documents' en
ln `into a MIF file and then import the MIF file into a FrameMaker' en
ln `document with the graphics and document formats intact.' en
ep
                                    # End of MIF File
```

Bar chart example

This example shows a bar chart and the MIF file that describes it. This example is in the file `barchart.mif`.

To draw the bar chart, you open or import the MIF file in FrameMaker. Normally, you would create an anchored frame in a document, select the frame, and then import this file. The MIF statements to describe the bar chart can be created by a database publishing application that uses the values in a database to determine the size of the bars.



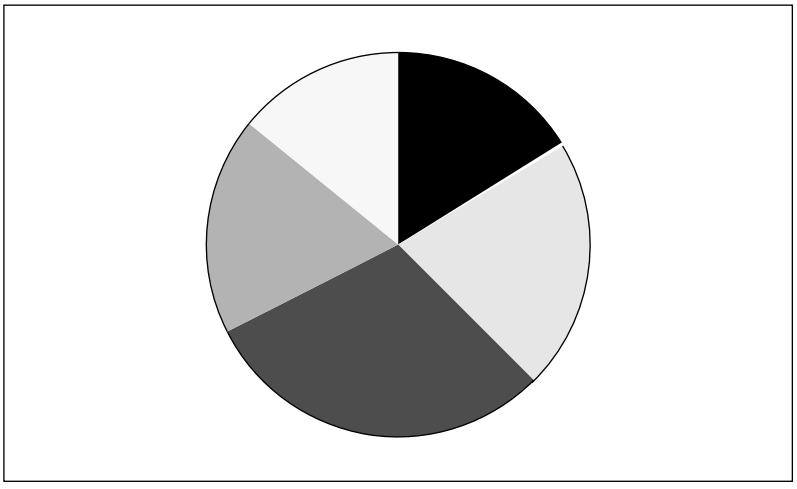

```
<MIFFile 2015>                      # Generated by SomeChartPack 1.4; identifies this
                                    # as a MIF file.
                                    # Chart title, in a text line.
                                    # All objects in the chart are grouped, so they have the
same
                                    # Group ID.
<TextLine <GroupID 1>
    <Font <FFamily `Times'> <FSize 14> <FPlain Yes> <FBold Yes>
    <FDX 0> <FDY 0> <FDAX 0> <FNoAdvance No>
    >
    <TLOrigin 1.85" 0.21"> <TLAlignment Center> <String `Market Shares'>
>                                   # end of TextLine
                                    # Boxes for Brand F and Brand I legends.
<Rectangle <GroupID 1>
    <Fill 1>
    <ShapeRect  1.36" 0.33" 0.38" 0.13">
>
<Rectangle <GroupID 1>
    <Fill 4>
    <ShapeRect  1.36" 0.54" 0.38" 0.13">
>
                                    # Text lines for Brand F and Brand I legends.
<TextLine <GroupID 1>
    <Font <FSize 12> <FPlain Yes>>
    <TLOrigin  1.80" 0.46"> <TLAlignment Left> <String `Brand F'>
>
                                    # Second text line inherits the current font from the
                                    # preceding text line.
<TextLine <GroupID 1>
    <TLOrigin  1.80" 0.67"> <TLAlignment Left> <String `Brand I'>
>
                                    # Reset the current pen pattern and pen width for
subsequent
                                    # objects.
<Pen 0>
<PenWidth 0.500>
```

```
                                    # Axes for the chart.
<PolyLine <GroupID 1>  <Fill 15>
    <NumPoints 3> <Point  0.60" 0.08"> <Point  0.60" 2.35"> <Point  3.10" 2.35">
>
                                    # Tick marks along the y axis.
<PolyLine <GroupID 1>
    <NumPoints 2> <Point  0.60" 1.83"> <Point  0.47" 1.83">
>
<PolyLine <GroupID 1>
    <NumPoints 2> <Point  0.60" 1.33"> <Point  0.47" 1.33">
>
<PolyLine <GroupID 1>
    <NumPoints 2> <Point  0.60" 0.83"> <Point  0.47" 0.83">
>
<PolyLine <GroupID 1>
    <NumPoints 2> <Point  0.60" 0.33"> <Point  0.47" 0.33">
>
                                    # X-axis labels.
<TextLine <GroupID 1>
    <TLOrigin  1.08" 2.51"> <TLAlignment Center> <String ` 1986 '>
>
<TextLine <GroupID 1>
    <TLOrigin  1.58" 2.51"> <TLAlignment Center> <String ` 1987 '>
>
<TextLine <GroupID 1>
    <TLOrigin  2.08" 2.51"> <TLAlignment Center> <String ` 1988 '>
>
<TextLine <GroupID 1>
    <TLOrigin  2.58" 2.51"> <TLAlignment Center> <String ` 1989 '>
>
                                    # Y-axis labels.
<TextLine <GroupID 1>
    <TLOrigin  0.46" 1.92"> <TLAlignment Right>  <String ` 25% '>
>
<TextLine <GroupID 1>
    <TLOrigin  0.46" 1.42"> <TLAlignment Right>  <String ` 50% '>
>
<TextLine <GroupID 1>
    <TLOrigin  0.46" 0.92"> <TLAlignment Right>  <String ` 75% '>
>
<TextLine <GroupID 1>
    <TLOrigin  0.46" 0.42"> <TLAlignment Right>  <String ` 100% '>
>
                                    # Draw all the gray bars first, since they have the same
fill.
                                    # Set the fill for the first bar; the others inherit the
fill
                                    # pattern.
<Rectangle <GroupID 1>
    <Fill 4>
    <ShapeRect  0.97" 1.10" 0.13" 1.25">
>
<Rectangle <GroupID 1>
    <ShapeRect  1.47" 1.47" 0.13" 0.88">
>
<Rectangle <GroupID 1>
    <ShapeRect  1.97" 1.72" 0.13" 0.63">
>
<Rectangle <GroupID 1>
    <ShapeRect  2.47" 1.97" 0.13" 0.38">
>
```

```
                                    # Now draw all the black bars, since they have the same
fill.
                                    # Set the fill for the first bar; the others inherit the
fill
                                    # pattern.
<Rectangle <GroupID 1>
    <Fill 1>
    <ShapeRect  1.10" 1.97" 0.13" 0.38">
>
<Rectangle <GroupID 1>
    <ShapeRect  1.60" 1.72" 0.13" 0.63">
>
<Rectangle <GroupID 1>
    <ShapeRect  2.10" 1.22" 0.13" 1.13">
>
<Rectangle <GroupID 1>
    <ShapeRect  2.60" 0.85" 0.13" 1.50">
>
                                    # Define the group for all the objects to make the chart
easier
                                    # to
                                    # manipulate after it's imported into a FrameMaker
document.
<Group <ID 1>
>
```

Pie chart example

When the MIF in this sample is imported into a page or graphic frame in a document, FrameMaker centers the chart in the page or graphic frame. This example is in the file `piechart.mif`.



```
<MIFFile 2015>                      # Generated by xyzgrapher 3.5; identifies this as a
                                    # MIF file.
                                    # All dimensions are in points.
<Units Upt >
                                    # Set the current pen pattern, width, and fill pattern.
<Pen 0>
```

```
<PenWidth .5>
<Fill 0>
                                    # Draw the black arc.
                                    # All arcs are part of the same circle, so they have the
same
                                    # ArcRect.
                                    # All objects in the chart are grouped, so they have the
same
                                    # Group ID.
<Arc <GroupID 1>
    <ArcRect  12 11 144 144 > <ArcTheta 0> <ArcDTheta 58>
>
                                    # Continue clockwise around the chart.
<Arc <Fill 5> <GroupID 1>
    <ArcRect  12 11 144 144 > <ArcTheta 58> <ArcDTheta 77>
>
<Arc <Fill 2> <GroupID 1>
    <ArcRect  12 11 144 144 > <ArcTheta 135> <ArcDTheta 108>
>
<Arc <Fill 4> <GroupID 1>
    <ArcRect  12 11 144 144 > <ArcTheta 243> <ArcDTheta 66>
>
<Arc <Fill 6> <GroupID 1>
    <ArcRect  12 11 144 144 > <ArcTheta 309> <ArcDTheta 51>
>
                                    # Define the group for all the objects to make the chart
easier
                                    # to manipulate after it's imported into a FrameMaker
                                    # document.
<Group <ID 1> >
```

Custom dashed lines

FrameMaker provides eight predefined dashed line options. You can define a custom pattern for dashed lines by using the `DashedPattern` statement within an *`Object`* statement. This example is in the file `custdash.mif`.

```
<MIFFile 2015>
                                    # This is a sparse dot-dash line.
<PolyLine
 <Pen 0>
 <Fill 15>
 <PenWidth 4pt>
 <ObColor `Black'>
 <DashedPattern
  <DashedStyle Dashed>
  <NumSegments 4>
  <DashSegment 10pt>
  <DashSegment 10pt>
  <DashSegment 0.5pt>
  <DashSegment 10pt>
 >                                  # end of DashedPattern
 <HeadCap Round>
 <TailCap Round>
 <NumPoints 2>
 <Point 1.0" 1">
 <Point 7.5" 1">
>                                   # end of PolyLine
                                    # This is a very sparse dotted line.
```

```
<PolyLine
 <DashedPattern
  <DashedStyle Dashed>
  <NumSegments 2>
  <DashSegment 0.5pt>
  <DashSegment 20pt>
 >                                  # end of DashedPattern
                                    # The polyline inherits round head caps and tail caps
from
                                    # the previous PolyLine statement.
 <NumPoints 2>
 <Point 1.0" 2">
 <Point 7.5" 2">
>                                   # end of PolyLine
                                    # This is a wild one!
<PolyLine
 <DashedPattern
  <DashedStyle Dashed>
  <NumSegments 8>
  <DashSegment  4pt>                    # solid
  <DashSegment  8pt>
  <DashSegment 12pt>                    # solid
  <DashSegment 16pt>
  <DashSegment 20pt>                    # solid
  <DashSegment 24pt>
  <DashSegment 20pt>                    # solid
  <DashSegment 16pt>
  <DashSegment 12pt>                    # solid
  <DashSegment  8pt>
 >                                  # end of DashedPattern
 <HeadCap Butt>
 <TailCap Butt>
 <NumPoints 2>
 <Point 1.0" 3">
 <Point 7.5" 3">
>                                   # end of PolyLine
                                    # This one has a missing DashSegment statement, so the
first
                                    # 10-point segment is repeated with a default gap of 10
points.
<PolyLine
 <DashedPattern
  <DashedStyle Dashed>
                                    # Missing NumSegments.
  <DashSegment 10pt>
                                    # Missing a second DashSegment.
 >
                                    # This polyline inherits the butt cap and tail style
                                    # from the previous PolyLine statement.
 <NumPoints 2>
 <Point 1.0" 4">
 <Point 7.5" 4">
>                                   # end PolyLine
                                    # This one is a really dense dotted line.
<PolyLine
 <DashedPattern
  <DashedStyle Dashed>
  <DashSegment 1pt>
  <DashSegment 1pt>
 >
                                    # This polyline also inherits the butt cap and tail style
```

```
                                    # from the previous PolyLine statement.
 <PenWidth 1pt>
 <NumPoints 2>
 <Point 1.0" 5">
 <Point 7.5" 5">
>                                   # end PolyLine
```

When you've defined a custom dashed line style in one FrameMaker document, you can easily copy and paste the custom style into another document by pressing Shift and choosing Pick Up Object Properties from the Graphics menu. For more information, see your user's manual.

Table examples

You can use MIF to create a table or to update a few values in an existing table.

Creating an entire table

This example shows a table and the MIF file that describes it. This table is in the sample file `stocktbl.mif`. The widths of columns is calculated using MIF statements that are only for input filters. Rather than specifying an exact width for each column, the table uses the substatement `TblColumnWidthA` for two of the columns to specify that the column width is determined by the width of a particular cell.

Column widths are further affected by the `EqualizeWidths` statement, which sets the columns to the width of the widest column within the limits specified by the `TblColumn` substatements. As you examine this example, note how the column width statements interact: the column widths are originally set by the applied table format from the Table Catalog. The `TblFormat` statement then specifies how this table instance's column properties override those in the default format. The `EqualizeWidths` statement further overrides the format established by `TblFormat`.

Table 2: StockWatch

Mining and Metal	10/31/90 Close	Weekly % Change
Ace Aluminum	$24.00	-3.50
Streck Metals	$27.25	+2.75
Linbrech Alloys	$63.75	-2.50

```
<MIFFile 2015>                      # Generated by StockWatcher; identifies this as a
                                    # MIF file.
<Tbls
 <Tbl
  <TblID 1>                         # This table's ID is 1.
  <TblFormat
   <TblTag `Format A'>
                                    # Forces a lookup in the Table Catalog with the following
                                    # exceptions:
   <TblColumn
    <TblColumnNum 0>
                                    # Shrink-wrap the first column so it's between 0 and 2
inches
                                    # wide.
    <TblColumnWidthA 0 2">
   >
```

```
   <TblColumn
    <TblColumnNum 1>
                                  # Make 2nd column 1 inch wide. This establishes a minimum
                                  # width for the columns.
    <TblColumnWidth 1">
   >
   <TblColumn
    <TblColumnNum 2>
                                  # Shrink-wrap the third column to the width of its
heading
                                  # cell.
                                  # See CellAffectsColumnWidthA statement below.
    <TblColumnWidthA 0 2">
   >
  >                               # end of TblFormat
                                  # The table instance has three columns.
  <TblNumColumns 3>
  <EqualizeWidths
                                  # Make the width of the second and third columns equal to
                                  # the larger of the two.  However, the columns cannot be
wider
                                  # than 2 inches or narrower than 1 inch.
   <TblColumnNum 1>
   <TblColumnNum 2>
  >                               # end of EqualizeColWidth
  <TblTitle
   <TblTitleContent
    <Para
                                  # Forces lookup in Paragraph Catalog.
     <PgfTag `TableTitle'>
      <ParaLine
       <String `StockWatch'>
      >                           # end of ParaLine
    >                             # end of Para
   >                              # end of TblTitleContent
  >                               # end of TblTitle
  <TblH                           # The heading.
   <Row                           # The heading row.
    <Cell <CellContent <Para               # Cell in column 0.
     <PgfTag `CellHeading'>                       # Forces lookup in Paragraph Catalog.
     <ParaLine <String `Mining and Metal'>>>>
    >                             # end of Cell
    <Cell <CellContent <Para                  # Cell in column 1
     <PgfTag `CellHeading'>                   # Forces lookup in Paragraph Catalog.
     <ParaLine <String `10/31/90 Close'>>>>
    >                             # end of Cell
    <Cell <CellContent <Para               # Cell in column 2
     <PgfTag `CellHeading'>                # Forces lookup in Paragraph Catalog.
     <ParaLine <String `Weekly %'> <Char HardReturn>>
     <ParaLine <String `Change'>>>>
                                  # For shrink-wrap.
     <CellAffectsColumnWidthA Yes>
    >                             # end of Cell
   >                              # end of Row
  >                               # end of TblH
  <TblBody                        # The body.
   <Row                           # The first body row.
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                   # Forces lookup in Paragraph Catalog.
     <ParaLine <String `Ace Aluminum'>>>>
    >                             # end of Cell
```

```
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `$24.00'>>>>
    >                               # end of Cell
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `-3.50'>>>>
    >                               # end of Cell
   >                                # end of Row
   <Row                             # The second body row.
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `Streck Metals'>>>>
    >                               # end of Cell
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `$27.25'>>>>
    >                               # end of Cell
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `+2.75'>>>>
    >                               # end of Cell
   >                                # end of Row
   <Row                             # The third body row
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `Linbrech Alloys'>>>>
    >                               # end of Cell
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `$63.75'>>>>
    >                               # end of Cell
    <Cell <CellContent <Para
     <PgfTag `CellBody'>                    # Forces lookup in Paragraph Catalog.
     <ParaLine <String `-2.50'>>>>
    >                               # end of Cell
   >                                # end of Row
  >                                 # end of TblBody
 >                                  # end of Tbl
>                                   # end of Tbls
<TextFlow <Para
 <PgfTag Body>
 <ParaLine <ATbl 1>>                    # Reference to table ID 1.>>
```

Updating several values in a table

You can update several values in a table (or elsewhere in a document) by importing a MIF file.

To update a table, insert a table in a FrameMaker document and create user variables for the values you want to update (see your user's manual); then insert the variables in the table where you want them.

To change the values of the variables, create a MIF file with new variable definitions. You can create MIF variable definitions from sources such as records in a database, values in a spreadsheet, or data gathered from measurement equipment. For example, the following MIF file defines two variables:

```
<MIFFile 2015>
<VariableFormats
    <VariableFormat
    <VariableName `90 Revenue'>
    <VariableDef `2,342,165'>
>
```

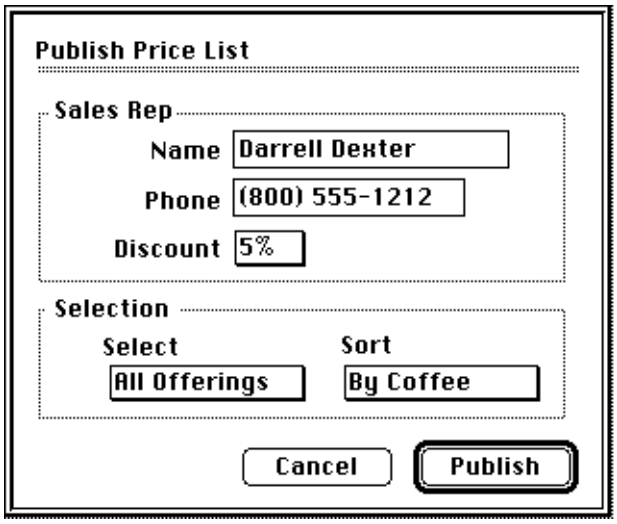
```
<VariableFormat
    <VariableName `91 Revenue'>
    <VariableDef `3,145,365'>
>>
```

When you import the MIF file into the document that contains the table, FrameMaker updates the variables in the table.

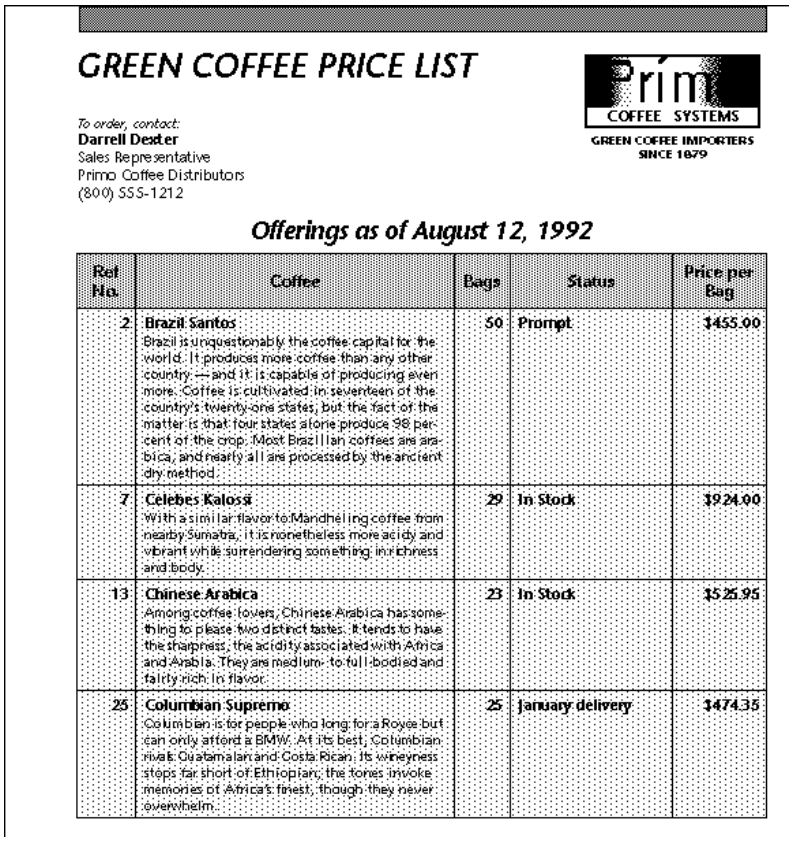
Database publishing

This database publishing example shows how to use the data storage and manipulation capabilities of a database and the formatting capabilities of FrameMaker through MIF.

In this example, inventory information for a coffee distributor is stored in a database. Database fields contain a reference number, the type of coffee, the number of bags in inventory, the current inventory status, and the price per bag. A sales representative creates an up-to-date report on the coffee inventory by using a customized dialog box in the database application to select the category of information and sort order:



When the sales representative clicks Publish, a database procedure scans the database, retrieves the requested information, and writes a MIF file that contains all of the information in a fully formatted document. The final document looks like this:



GREEN COFFEE PRICE LIST

Primo COFFEE SYSTEMS
GREEN COFFEE IMPORTERS SINCE 1879

To order, contact:
Darrell Dexter
Sales Representative
Primo Coffee Distributors
(800) 555-1212

Offerings as of August 12, 1992

Ref No.	Coffee	Bags	Status	Price per Bag
2	**Brazil Santos** Brazil is unquestionably the coffee capital for the world. It produces more coffee than any other country — and it is capable of producing even more. Coffee is cultivated in seventeen of the country's twenty-one states, but the fact of the matter is that four states alone produce 98 percent of the crop. Most Brazilian coffees are arabica, and nearly all are processed by the ancient dry method.	50	Prompt	$455.00
7	**Celebes Kalossi** With a similar flavor to Mandheling coffee from nearby Sumatra, it is nonetheless more acidy and vibrant while surrendering something in richness and body.	29	In Stock	$924.00
13	**Chinese Arabica** Among coffee lovers, Chinese Arabica has something to please two distinct tastes. It tends to have the sharpness, the acidity associated with Africa and Arabia. They are medium- to full-bodied and fairly rich in flavor.	23	In Stock	$525.95
25	**Columbian Supremo** Columbian is for people who long for a Royce but can only afford a BMW. At its best, Columbian rivals Guatamalan and Costa Rican. Its wineyness stops far short of Ethiopian; the tones invoke memories of Africa's finest, though they never overwhelm.	25	January delivery	$474.35

The data from the database is published as a FrameMaker table. The database procedure makes one pass through the records in the database and writes the contents of each record in a row of the table. The procedure then creates a `TextFlow` statement that contains the text that appears above the table and creates an `ATbl` statement to refer to the table instance.

You can set up a report generator like the previous example by following these general steps:

1 Create the template for the final report in FrameMaker. Design the master pages and body pages for the document and create paragraph and character formats. You can include graphics (such as a company logo) on the master page.

2 Create a table format for the report. Specify the table position, column format, shading, and title format. Store the format in the Table Catalog.

3 When the document has the appearance you want, save it as a MIF file.

4 Edit the MIF file to create a MIF template that you can include in your generated MIF file (see "Including template files" on page 45). The MIF template used for this example is in the sample file `coffee.mif`.

5 Use your database to create any custom dialog boxes or report-generating procedures.

6 Create a database query, or procedure, that extracts data from the database and writes it out into a MIF file. Use a MIF `include` statement to include the document template in the new document.

The database user can now open a fully formatted report.

The code for the procedure that extracts information from the database and outputs the MIF strings is shown in this appendix. This procedure is written in the ACIUS 4th DIMENSION command language. You could use any database query language to perform the same task.

The procedure does the following:

7 Creates a new document.

8 Sends the `MIFFile` identification line.

9 Uses `include` to read in the formatting information stored in the template `coffee.mif`.

10 Sends the MIF statements to create a table instance.

11 In each body cell, sends a field that includes the information extracted from the database.

12 Creates a text flow that uses the `TextRectID` from the empty body page in the `coffee.mif` template.

13 Includes the `Atbl` statement that places the table instance in the document text flow.

14 Closes the document.

In the following example, database commands are shown like this: **SEND PACKET**. Comments are preceded by a single back quote (`). Local variables are preceded by a dollar sign ($).

`This procedure first gets the information entered by the user and stores it in local variables:

```
  `   $1 = Name of sales representative
  `   $2 = Phone number
  `   $3 = Discount
CR:=char(13) ` carriage return character
DQ:=char(34) ` double quotation mark character
C_TIME(vDoc)
CLOSE DOCUMENT(vDoc)
vDoc:=Create document("")
vDisc:=1-(Num($3»)/100)
  `Send header.
SEND PACKET(vDoc;"<MIFFile 2015> #Generated by 4th Dimension for Version 7.0 of
FrameMaker"+CR)
  `Read in the MIF template for the report.
SEND PACKET(vDoc;"include (coffee.mif)"+CR)
  `Generate table.
```

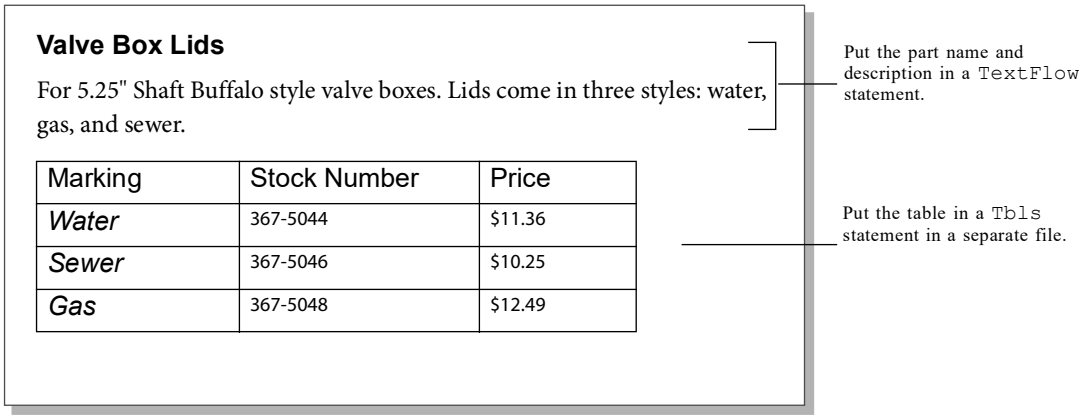
```
SEND PACKET(vDoc;"<Tbls <Tbl <TblID 2> <TblFormat <TblTag `Format A'>>"+CR)
        SEND PACKET(vDoc;"<TblNumColumns 5> <TblColumnWidth .6"+DQ+">"+CR)
        SEND PACKET(vDoc;"<TblColumnWidth 3.25"+DQ+">"+CR)
        SEND PACKET(vDoc;"<TblColumnWidth .5"+DQ+">"+CR)
        SEND PACKET(vDoc;"<TblColumnWidth 1.7"+DQ+">"+CR)
        SEND PACKET(vDoc;"<TblColumnWidth 1.0"+DQ+">"+CR)
        SEND PACKET(vDoc;"<TblTitle"+CR)
        SEND PACKET(vDoc;"<TblTitleContent"+CR)
        SEND PACKET(vDoc;"<Para <PgfTag `TableTitle'>"+CR)
        SEND PACKET(vDoc;"<ParaLine <String `Offerings as of "+String(Current
        date;5)+"'>>>>>"+CR)
          `Table Heading Row.
        SEND PACKET(vDoc;"<TblH <Row <RowMaxHeight 14.0"+DQ+"> "+CR)
        SEND PACKET(vDoc;"<Cell <CellContent <Para <PgfTag `CellHeading'>"+CR)
        SEND PACKET(vDoc;"<ParaLine <String `Ref No.'>>>>>"+CR)
        SEND PACKET(vDoc;"<Cell <CellContent <Para <PgfTag `CellHeading'>"+CR)
        SEND PACKET(vDoc;"<ParaLine <String `Coffee'>>>>>"+CR)
        SEND PACKET(vDoc;"<Cell <CellContent <Para <PgfTag `CellHeading'>"+CR)
        SEND PACKET(vDoc;"<ParaLine <String `Bags'>>>>>"+CR)
        SEND PACKET(vDoc;"<Cell <CellContent <Para <PgfTag `CellHeading'>"+CR)
        SEND PACKET(vDoc;"<ParaLine <String `Status'>>>>>"+CR)
        SEND PACKET(vDoc;"<Cell <CellContent <Para <PgfTag `CellHeading'>"+CR)
          `Retail and Discount prices are conditional.
        SEND PACKET(vDoc;"<ParaLine <Conditional <InCondition `Retail'>>"+CR)
        SEND PACKET(vDoc;"<String `Price per Bag'>"+CR)
        SEND PACKET(vDoc;"<Conditional <InCondition `Discount'>> <String `Discount
        Price'>"+CR)
        SEND PACKET(vDoc;"<Unconditional> >>>>>>"+CR)
          `Table Body.
        FIRST RECORD([Inventory])
        SEND PACKET(vDoc;"<TblBody"+CR)
        For ($n;1;Records in selection([Inventory])
          `Change shading of row depending on inventory status.
        If ([Inventory]Status="In stock")
            vFill:="<CellFill 6> <CellColor `Green'>"
          Else
            vFill:=" <CellFill 6> <CellColor `Red'>"
          End if
          `Compute discount price.
          vDiscPrice:=[Inventory]Price per Bag*vDisc
          RELATE ONE([Inventory]Name)
          SEND PACKET(vDoc;"<Row <RowMaxHeight 14.0"+DQ+">"+CR)
          SEND PACKET(vDoc;"<Cell "+vFill+" <CellContent <Para <PgfTag
        `Number'>"+CR)
          SEND PACKET(vDoc;"<ParaLine <String `"+String([Inventory]Ref
        Number;"###")+"'>>>>>"+CR)
          SEND PACKET(vDoc;"<Cell "+vFill+" <CellContent <Para <PgfTag `Body'>"+CR)
          SEND PACKET(vDoc;"<ParaLine <String `"+[Inventory]Name+"'>>>"+CR)
          SEND PACKET(vDoc;"<Para <PgfTag `CellBody'>"+CR)
          SEND PACKET(vDoc;"<ParaLine <String `"+[Beans]Description+"'>>>>>"+CR)
          SEND PACKET(vDoc;"<Cell "+vFill+" <CellContent <Para <PgfTag
```

```
`Number'>"+CR)
  SEND PACKET(vDoc;"<ParaLine <String
`"+String([Inventory]Bags;"###")+"'>>>>>"+CR)
  SEND PACKET(vDoc;"<Cell "+vFill+" <CellContent <Para <PgfTag `Body'>"+CR)
  SEND PACKET(vDoc;"<ParaLine <String `"+[Inventory]Status+"'>>>>>"+CR)
  SEND PACKET(vDoc;"<Cell "+vFill+" <CellContent <Para <PgfTag
`Number'>"+CR)
  SEND PACKET(vDoc;"<ParaLine <Conditional <InCondition `Retail'>>"+CR)
  SEND PACKET(vDoc;"<String `"+String([Inventory]Price per Bag;"$#,###.00")+"'>")
  SEND PACKET(vDoc;"<Conditional <InCondition `Discount'>>"+CR)
  SEND PACKET(vDoc;"<String `"+String(vDiscPrice;"$###,###.00")+"'> "+CR)
  SEND PACKET(vDoc;"<Unconditional> >>>>>"+CR)
  MESSAGE("Generating MIF for "+[Inventory]Name+", Status:
"+[Inventory]Status+".")
  NEXT RECORD([Inventory])
End for
SEND PACKET(vDoc;">>>"+CR)  `End of table.
  `Body of page.
SEND PACKET(vDoc;"<TextFlow <TFTag `A'> <TFAutoConnect Yes>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Heading'> <ParaLine <TextRectID 8>"+CR)
SEND PACKET(vDoc;"<String `GREEN COFFEE PRICE LIST'> <AFrame
1>>>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Prepared'> <ParaLine <String `To order,
contact:'>>>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Body'> <ParaLine <String
`"+$1»+"'>>>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Body2'>"+CR)
SEND PACKET(vDoc;"<ParaLine <String `Sales Representative'>>>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Body2'>"+CR)
SEND PACKET(vDoc;"<ParaLine <String `Primo Coffee Distributors'>>>"+CR)
SEND PACKET(vDoc;"<Para <PgfTag `Body2'> <ParaLine "+CR)
SEND PACKET(vDoc;"<String `"+String(Num($2»);"(###) ###-####")+"'>"+CR)
SEND PACKET(vDoc;"<ATbl 2> >>>"+CR)  `Send the anchor for the table
CLOSE DOCUMENT(vDoc)
ALERT("Your MIF file is awaiting your attention.")
```

Creating several tables

The previous example illustrates how to use a database to create one table instance. Both the `Tbls` and the `TextFlow` statements are written to a single text file. This approach, however, is limited to this simple case. If the document contains several tables, it may be more convenient to use the database to write the `Tbls` statement to a separate file and then use a MIF `include` statement to read the file into FrameMaker.

For example, suppose you need to publish a parts catalog. Each part has a name, a description, and a table that gives pricing information. A typical record looks like this:



In the database, all the information about each part is associated with its record. Due to the structure of MIF, however, the information must appear in different portions of the MIF file: the part name and description belong in the `TextFlow` statement, while the table belongs in the `Tbls` statement. To accomplish this, you can make the following modifications to the design of the database procedure shown in the previous example.

- At the beginning of the procedure, create two text files—one for the main MIF file that will contain the MIF file identification line and the main text flow and the other for the `Tbls` statement.
- Use a second `include` statement to read in the `Tbls` statement
- As your procedure passes through each record, write the data that belongs in the `TextFlow` statement in the main text file and write the table data to the `Tbls` file.

If you are using 4th Dimension, the procedure should have the following statements:

```
vDoc:=CREATE DOCUMENT ("") `Prompts user to name main file.
vTbls:=CREATE DOCUMENT (Tbls.mif) `Hard codes name of include file.
SEND PACKET (vDoc;"<MIFFile 2015> #File ID")
SEND PACKET (vDoc;"include (template.mif")
SEND PACKET (vDoc;"include (Tbls.mif")
```

As you process the records, you write the table data to the second include file by referring to the vTbls variable in a SEND PACKET command. For example:

```
SEND PACKET (vTbls; "<Cell <CellContent"+CR)
```

The main MIF file would have the following components:

```
<MIFFile 2015>                          # File ID
include (template.mif)                  # MIF template
include (Tbls.mif)                      # Table instances, created by
# the database
<TextFlow                               # Main text flow
...
>                                       # end of text flow
```

When FrameMaker opens the main MIF file, it will use the two `include` statements to place the data and template information in the required order.

Creating anchored frames

You can extend the technique of writing separate MIF files to handle both tables and graphics. Like table instances, anchored frame instances must appear in the MIF file prior to the `TextFlow` statement. If each record contains a graphic or a reference to a graphics file on disk, you would create a separate text file called `AFrames.mif` for only the `AFrames` statement. Using the technique described in the previous section, you would insert the code for the tables in the `Tbls.mif` file, the graphics in the `AFrames.mif` file, and the main text flow in the main text file. You use an `include` statement to read in the `AFrames.mif` file.

Note: *Remember to assign unique ID numbers in the* `TblID` *statement for each table and the* `ID` *statement for each frame.*

Chapter 9: MIF Messages

When the MIF interpreter reads a MIF file, it might detect errors such as unexpected character sequences. In UNIX versions, the MIF interpreter displays messages in a console window. In the Windows versions, you must turn on Show File Translation Errors in the Preferences dialog box to display messages in a window (a console window in the Windows version). If the MIF interpreter finds an error, it continues to process the MIF file and reads as much of the document as possible.

General form for MIF messages

The general form of all MIF messages is:

`MIF: Line LineNum: Message`

The `LineNum` may be approximate because it represents the absolute line number in the file after all macros in the file have been expanded. In addition, if you open the MIF file in Adobe® FrameMaker®, lines are wrapped and the line numbers may change.

The `Message` portion consists of one of the messages in the following table. (Italicized words/characters (for example, *n*) indicate variable words or values in a message.)

List of MIF messages

The tables in this section lists the MIF messages produced by the MIF interpreter and describes their meanings.

This message	Means
--- Skipping these chars: ...*(MIF statements)*... ---------- Done skipping.	The MIF file contains a syntax error or a MIF statement not supported in this version of FrameMaker. FrameMaker ignores all MIF statements contained within the erroneous or unsupported MIF statement. The ignored MIF statements are listed in the error message.
A footnote cannot contain another footnote.	One footnote in the MIF file is embedded in another.
Bad parameter: *parameter*.	The MIF file contains a syntax error.
Cannot connect to TRNext ID *n*.	The text frame ID specified in a `TRNext` statement has no corresponding defined text frame.
Cannot find anchored frame *n*.	The graphic frame ID specified in an `AFrames` statement has no corresponding defined graphic frame.
Cannot find footnote *n*.	The footnote ID specified in a `FNote` statement has no corresponding defined footnote.
Cannot find table ID *n*.	MIF cannot match `<ATbl x>` with an earlier `<Tbl <TblID x>>` statement.
Cannot find text frame ID *n*.	The text frame ID specified in a `TextRectID` statement has no corresponding defined text frame.
Cannot open *filename*.	Make sure that the file exists and that you have read access to it; then try again.

This message	Means
Cannot store inset's facets.	The MIF file contains a graphic inset, but the MIF interpreter can't store the graphic inset in the document. There might be an error in the MIF syntax, or there might not be enough temporary disk space available. In UNIX versions, try to increase the space available in your home directory or the `/usr/tmp` directory and try again. In the Windows versions, try quitting other applications and closing other open windows; then start FrameMaker again.
Char out of range: *character_value*.	A character in a `Char` statement or a character expressed using `\x` in a string is out of range.
Condition settings must not change between <XRef> and <XRefEnd>.	You cannot change a condition tag setting in the middle of a cross-reference. Make sure the entire cross-reference is contained in one condition setting.
DashedPattern statement has no DashedSegment statements.	A `DashedPattern` statement gives `DashedStyle` a value of `Dashed` but has no `DashedSegment` statements to define the dashed pattern.
Empty group: ID=*n*.	The group ID specified in a `Group` statement has no corresponding defined objects with a matching group ID.
Expected comma/identifier/left parenthesis/right parenthesis/right quote.	The MIF file contains a syntax error.
Following <TabStop> statements will determine actual number of tabs.	The `PgfNumTabs` statement is present in MIF for use by other programs that read MIF files; it is not used by the MIF interpreter. When the MIF interpreter reads a MIF file, it counts the number of `TabStop` statements to determine the number of tabs stops in a paragraph.
Frames are nested too deeply (over 10); skipping statement.	There are too many nested frames. The maximum nesting depth is 10.
Graphic frame has an invalid <Angle> attribute.	An invalid value is specified by the `Angle` statement for a graphic frame.
Insufficient memory!	FrameMaker cannot allocate enough memory for one of its work buffers. In UNIX versions, try to free some swap space and restart FrameMaker. In the Windows versions, try quitting other applications and closing other open windows; then start FrameMaker again.
Invalid opcode: *op_code*.	The MIF file contains a syntax error.
Macro/IncludeFile nesting too deep.	The define or include statements specify too many nested levels of statements.
Missing dimension.	A necessary dimension value was not found in a MIF statement.
No name was given for the cross-reference format: *format_definition*.	The `XRefName` statement is not specified for a cross-reference format.
No name was given for the variable definition: *variable_definition*.	The `VariableName` statement is not specified for a variable.
Object ignored; must come before <TextFlow> statements.	All object statements must come before the first `TextFlow` statement in a MIF file.
Processing opcode *op_code*.	FrameMaker is currently processing the specified opcode.
Skipped '*string*'.	The MIF file contains a syntax error.

This message	Means
String too long (over 255 or 1023 characters); overflow ignored.	The maximum length for most <UserString> strings is 1023 characters. The maximum length for all other strings is 255 characters.
Structured MIF statement ignored.	This FrameMaker is set to use the unstructured program interface, and so it does not support structured MIF statements.
Syntax error in <MathFullForm> statement.	The MIF file contains a syntax error in a `MathFullForm` statement.
Unable to start new object.	FrameMaker cannot allocate memory for a new object. In UNIX versions, try to free some swap space and restart FrameMaker. In the Windows versions, try quitting other applications and closing other open windows; then start FrameMaker again.
Unable to store marker.	The marker table is full. In UNIX versions, FrameMaker is probably running out of swap space. Try to free some swap space and restart FrameMaker. In the Windows versions, try quitting other applications and closing other open windows; then start FrameMaker again.
Unbalanced right angle bracket.	A right angle bracket (>) was found that has no corresponding left angle bracket (<).
Unexpected opcode.	A statement was found in a context where it is not valid (for example, an `FFamily` statement in a `Document` statement).
Unexpected right angle bracket.	A right angle bracket (>) was found where a data value was expected or was found outside a statement.
Unknown font angle.	The requested font angle is not available.
Unknown font family.	The requested font family is not available.
Unknown font variation.	The requested font variation is not available.
Unknown font weight.	The requested font weight is not available.
Unknown PANTONE name: *string*.	The name specified in the `ColorPantoneValue` statement is not the name of a valid PANTONE color. Note: Adobe and Pantone have been working together to support your color needs. Pantone Matches are no longer supported in Adobe. Pantone color libraries currently preloaded in FrameMaker and FrameMaker Publishing Server will be phased out starting August 31, 2022 (with the exception of PANTONE+ CMYK Coated, PANTONE+ CMYK Uncoated, PANTONE+ Metallic Coated).
Value of *n* out of range *(m)*.	A statement's data value was too large or too small.
WARNING: Circular text flow was found and cut.	The MIF file defined a set of linked text frames resulting in a circular text flow. (The last text frame in the flow is linked to the first or to one in the middle.) The MIF interpreter attempted to solve the problem by disconnecting a text frame.
WARNING: Circular text flow. Don't use the document.	The MIF file defined a set of linked text frames resulting in a circular text flow. (The last text frame in the flow is linked to the first or to one in the middle.) The MIF interpreter was unable to solve the problem. A FrameMaker document file will open, but do not use it.

Chapter 10: MIF Compatibility

MIF files are compatible across versions. However, some MIF statements have changed in version 7.0 of Adobe® FrameMaker®. This appendix lists the MIF statements that are new or have changed in version 7.0 and describes how these statements are treated when an earlier version reads a 7.0 MIF file. The appendix also lists changes between versions 7.0 and 6.0, and between earlier version upgrades of FrameMaker. MIF statements are listed by feature.

In general, when previous versions of FrameMaker read new MIF statements, the new MIF statements are stripped out and ignored. For example, if version 4 of FrameMaker reads a new 7.0 MIF statement in a 7.0 MIF file, FrameMaker ignores the statement.

Changes between version 12.0 and 2015 release

This section describes changes to MIF syntax between versions 12.0 and FrameMaker (2015 release).

Language support

The `PgfLanguage` property of the `Pgf` statement now supports Arabic and Hebrew languages.

Numbering style

The following new numbering styles have been added:

- `IndicNumeric`
- `FarsiNumeric`
- `HebrewNumeric`
- `AbjadNumeric`
- `AlifbataNumeric`

These new numbering styles can be assigned at the paragraph level (`PgfNumFormat`), document level, or book level. At the document level, the numbering style is defined in the `Document` statement. The following properties of the `Document` statement can be configured to use the new numbering styles:

- `VolumeNumStyle`
- `ChapterNumStyle`
- `DPageNumStyle`
- `SectionNumStyle`
- `SubSectionNumStyle`
- `DFNoteNumStyle`
- `DTblFNoteNumStyle`

At the book level, the following properties of the `BookComponent` statement can be configured to use the new numbering styles:

- `VolumeNumStyle`
- `ChapterNumStyle`
- `SectionNumStyle`
- `SubSectionNumStyle`
- `PageNumStyle`
- `BFNoteNumStyle`
- `BTblFNoteNumStyle`

Document direction

The `DocDir` property defines the direction — left-to-right (LTR) or right-to-left (RTL), in which you can author your document. The objects that inherit their direction property from the `Document` would get affected if the `DocDir` property is changed.

Text flow direction

The `FlowDir` property controls the direction of the child objects that derive their direction from the flow. For example, a text frame can derive its direction from the text flow object.

You can also change the style of a text frame, in which case the `StyleCatalog` statement would contain a property named `TFrameDir`. This property controls the direction of all text frames created using the same style.

Paragraph direction

You can set the direction of a paragraph by using the `PgfDir` property. You can either change the direction of a single paragraph (`Para` statement) or a paragraph format (`Pgf` statement).

Table direction

You can set the direction of a table by using the `TblDir` property. You can either change the direction of a single table (`Tbl` statement) or a table format (`TblFormt` statement).

Text line Direction

The `TLDirection` property controls the direction in which the text line is drawn.

You can also change the style of a text line object, in which case the `StyleCatalog` statement contain a property named `TLineDir`. This property controls the direction of all text lines created using the same style.

Anchored frame direction

The `AnchorDirection` property controls the direction of individual anchored frame.

You can also change the style of an anchored frame, in which case the `StyleCatalog` statement would contain a property named `AFrameDir`. This property controls the direction of all anchored frames created using the same style.

Element direction

`ElemDir` property control the direction of an element in a structured document.

MathML style

You can change the style of the MathML equation by using the `MathMLStyleInline` and `MathMLApplyPgfStyle` properties. These properties allow a MathML equation to be inline with the enclosing paragraph's text or apply the formats of the enclosing paragraph.

Mini TOC

You can add a mini TOC to an unstructured document. The properties of `InlineComponentsInfo` statement defines the mini TOC properties.

Conditional table columns

Along with table rows, you can conditionalize table columns by using the `TableColumn` statement.

Changes between version 11.0 and 12.0

This section describes changes to MIF syntax between versions 11.0 and 12.0 of FrameMaker.

MathML

FrameMaker provides support for MathML, which is an XML application for representing mathematical notation. This support is provided through out-of-the-box integration with MathFlow Editor by Design Science. FrameMaker includes 30-day trial licenses of two MathFlow editors: Style Editor and Structure Editor.

In a MIF file, the `MathML` tag contains the various tags that hold MathML properties and data.

Paragraph box properties

You can set background color for paragraphs. In a MIF file, you can use the `PgfBoxColor` tag to set the background color of a paragraph.

Hotspot

A hotspot is an active area in a document that you can link to different areas of the document, to another document, or to a URL. You can apply hotspots to various objects, such as graphics, images, and anchored frames. In a MIF file, you can make an object a hotspot using the `IsHotspot` boolean tag. Using the `HotspotCmdStr` tag, you can specify the target URL or bookmark the user will go to after clicking the hotspot.

Object Style

You can save your frequently used object properties as a style. You can apply these object styles to various objects, such as images, anchored frames, and text frames for consistent size and appearance. For example, you can create and apply an object style to all the anchored frames in a document, or across documents, to make them of the same size.

In MIF files, the `StyleCatalog` tag contains the object styles and you can specify an object style using the `Style` tag.

Control Multimedia with links

You can insert links to interactively control embedded U3D (Universal 3D), FLV, and SWF objects in the PDF output. You can insert links to 3D and multimedia objects that control various aspects of these objects. You can also create a multimedia links table for the 3d\multimedia object of the type View, Parts, or Animation. For example, the multimedia links table of the type parts includes links that focus on different parts of the 3D\multimedia object.

In MIF files, you can specify support for multimedia links for an imported multimedia object using output but is not included in XML output. In a MIF document, you can turn on the banner text using the `DBannerTextOn` tag.

Line Numbers

Line numbers in FrameMaker files help you identify particular lines of content. Line numbers are set at a document

level (for a .fm file) and appear before each inserted line in a FrameMaker document. In a MIF document, you can enable line numbers using the `DLineNumShow` tag.

Dictionary Preferences

Using the dictionary preferences, you can specify Proximity or Hunspell dictionaries for Spelling and Hyphenation for various languages. In a MIF file, dictionary preferences are set in the `Dictionary` tag.

Changes between version 9.0 and 10.0

This section describes changes to MIF syntax between versions 9.0 and 10.0 of FrameMaker.

Text background color

In FrameMaker 10, you can add a background color for the paragraph and conditional text. In a MIF file, the background color for a paragraph tag is added using the `FBackgroundColor` tag and the background color for a conditional tag is added using the `CBackgroundColor` tag.

Track text edits

FrameMaker tracks the Windows/Unix username of the user who edits a document in track changes mode. FrameMaker also tracks the time of the edit. In a MIF document, this information is in the `DTrackChangesReviewerName`, `ReviewerName`, and `ReviewTimeInfo`.

Descriptive tags

FrameMaker displays the description of the elements in the element catalog. In a mif file, the `EDDescriptiveTag` tag contains the descriptive tag of an element and using a boolean tag `DShowElmentDescriptiveTags`, you can decide whether or not to display the element descriptions.

Custom catalogs

FrameMaker allows you to create custom catalogs of character formats, paragraph formats, and table formats. A mif document contains the boolean tags, `CustomPgfFlag`, `CustomFontFlag`, and `CustomTblFlag`, to control whether or not these custom catalogs exist in the document. For the custom catalogs, a mif document contains one tag each to signify the start of a custom catalog: `DCustomFontList`, `DCustomPgfList`, or `DCustomTblList`. The `DCustomFontTag`, `DCustomPgfTag`, and `DCustomTblTag` tags specify the names of the tags in the custom catalogs.

MIF syntax changes in FrameMaker 8

This section describes the MIF syntax changes in FrameMaker 8.

Filter By Attribute

Elements in a structured document can have one or more attributes associated with them. Using structured FrameMaker, you can filter a structured document based on the value of these attributes. The Filter by Attribute feature simplifies the task of filtering a structured document for complex output scenarios. You create a filter using the `DefAttrValuesCatalog`, `DefAttrValues`, `AttrCondExprCatalog`, and `AttrCondExpr` statements.

Track edited text

FrameMaker documents sent for review can be edited with the Track Text Edit feature enabled. In a MIF file, the Track Text Edit feature is enabled using the `DTrackChangesOn` Boolean statement.

Before you accept all text edits, you can choose to preview the final document with all the text edits incorporated in the document. Alternatively, you can preview the original document without the text edits incorporated in the document. You use the `DTrackChangesPreviewState` statement to preview the document.

Boolean condition expression

You can build Boolean expressions with complex combinations of condition tags and Boolean operators to generate conditional output.

In a MIF file, Boolean condition expressions are defined using a `BoolCond` statement. The `BoolCond` statement defines a new Boolean condition expression, which is used to evaluate the show/hide state of conditional text. This statement appears in the `BoolCondCatalog` statement.

New Book and Document related WebDAV statements

The `BookServerURL` and `BookServerState` MIF statements mark a book as managed content on the WebDAV-server. The `DocServerURL` and `DocServerState` MIF statements mark a document as managed content on the WebDAVserver.

Import graphics from HTTP file paths

You can specify an HTTP file path to import a graphic into a FrameMaker document either by copying or by reference.

The syntax of the `ImportObject` statement has been modified to provide this feature in FrameMaker. The ImportURL and ObjectInfo parameters have been included in the `ImportObject` MIF statement.

Changes between version 6.0 and 7.0

This section describes changes to MIF syntax between versions 6.0 and 7.0 of FrameMaker.

Changes to structured PDF

FrameMaker now includes attributes for graphic objects that are to be included when a document is saved as structured PDF. A graphic object can have an arbitrary number of attributes. Each attribute is stored in an `ObjectAttribute` statement. This statement contains one `Tag` statement and an arbitrary number of `Value` statements.

General XML support

In versions 7.0 and later, documents and books store general XML information such as XML version, encoding, and whether the XML is based on a DTD. This information is stored in the following statements:

Book statements	Document statements
BXmlDocType	DXmlDocType
BXmlEncoding	DXmlEncoding
BXmlFileEncoding	DXmlFileEncoding
BXmlPublicId	DXmlPublicId
BXmlStandAlone	DXmlStandAlone
BXmlStyleSheet	DXmlStyleSheet
BXmlSystemId	DXmlSystemId
BXmlUseBOM	DXmlUseBOM
BXmlVersion	DXmlVersion
BXmlWellFormed	DXmlWellFormed

XML Namespaces

In versions 7.0 and later, elements in structured FrameMaker documents now store namespace information. The `ENamespace` statement contains an arbitrary number of namespace declaration. Each namespace declaration consists of one `ENamespacePrefix` statement and one `ENamemespacePath` statement.

XMP job control packets

FrameMaker book and document files now store information to support XMP, the Adobe standard for collaboration and electronic job control. MIF stores XMP data in a series of encoded XMP statements that contain the data. You should not try to edit this data manually—FrameMaker generates the encoding when you save a file as MIF. This XMP data corresponds with the values of fields in the File Info dialog box. In MIF, this data is stored as sub-statements of `<DocFileInfo>` and `<BookFileInfo>`.

This XMP data contains the data that is stored in the `PDFDocInfo` and `PDFBookInfo` statements.

Changes between version 5.5 and 6.0

This section describes changes to MIF syntax between versions 5.5 and 6.0 of FrameMaker.

Saving documents and books as PDF

FrameMaker documents now store information to support Structured PDF. `DPDFStructure` is a new statement added to `Document` that specifies whether or not the document contains structure information to use when saving as PDF. `PgfPDFStructureLevel` has been added to the `Pgf` statement to assign a structure level to paragraph formats.

Books and documents can also include arbitrary fields of Document Info information. Documents use the `PDFDocInfo` statement, and books use `PDFBookInfo`.

To improve handling of bookmarks hypertext links within and across PDF files, FrameMaker now stores reference data within documents. `PgfReferenced` identifies each paragraph that is marked as a named destination; `ElementReferenced` similarly identified structure elements. If you like, you can specify that the Save As PDF function creates a named destination for every paragraph in the document; this is done via `FP_PDFDestsMarked` within the `Document` statement.

Books

Version 6.0 of FrameMaker has brought significant change to books. The book window now can display the filename of each book component, or a text snippet from the component's document. In MIF, `BDisplayText` determines which type of information to display.

A book can also be view-only; MIF now includes `BViewOnly`, `BViewOnlyWinBorders`, `BViewOnlyWinMenuBar`, `BViewOnlyPopup`, and `BViewOnlyNoOp` statements to express whether a book is view-only, and how it should appear.

Book Components

Book components store numbering properties to use when generating a book. The following table shows the new MIF statements for managing different types of numbering:

Volume	Chapter	Page	Footnote	Table Footnote
VolumeNumStart	ChapterNumStart	ContPageNum	BFNoteStartNum	BTblFNoteNumStyle
VolumeNumStyle	ChapterNumStyle	PageNumStart	BFNoteNumStyle	BTblFNoteLabels
VolumeNumText	ChapterNumText	PageNumStyle	BFNoteRestart	BTblFNoteCompute Method
VolNumCompute-Method	ChapterNumCompute-Method		BFNoteLabels	
			BFNoteCompute-Method	

Documents

Because there are new numbering properties for documents and books, documents now have new numbering statements. The following table shows the new MIF statements for managing different types of numbering in documents:

Volume	Chapter	Page	Footnote
VolumeNumStart	ChapterNumStart	ContPageNum	DFNoteComputeMethod
VolumeNumStyle	ChapterNumStyle	PageNumStart	
VolumeNumText	ChapterNumText	PageNumStyle	
VolNumComputeMethod	ChapterNumComputemethod		

Changes between version 5 and 5.5

This section describes changes to MIF syntax between versions 5 and 5.5 of FrameMaker.

Asian text processing

A section has been added to the *MIF Reference* to describe the new MIF statements that were added for Asian text in a document. See , "MIF Asian Text Processing Statements." for more information.

MIF file layout

A MIF file can now include a `CombinedFontCatalog` statement that contains `CombinedFontDefn` statements to define each combined font for the document. The `CombinedFontCatalog` statement must occur before the `Document` statement. For information about combined fonts, see "Combined Fonts" on page 214.

Control statements

A new control statement, `CharUnits`, has been added to express whether characters and line spacing is measured by points or by Q (the standard units of measurement for Japanese typography). The keywords for this statement are `CUpt` and `CUQ`.

Document statements

The `DPageNumStyle` and `DFNoteNumStyle` statements have new keywords to express Japanese footnote numbering formats. The new keywords are `ZenLCAlpha`, `ZenUCAlpha`, `KanjiNumeric`, `KanjiKazu`, and `BusinessKazu`.

`DTrapwiseCompatibility` is a new statement that determines whether generated PostScript will be optimized for the TrapWise application.

`DSuperscriptStretch`, `DSubscriptStretch`, and `DSmallCapsStretch` are new statements that specify the amount to stretch or compress superscript, subscript, or small caps text.

Color statements

MIF 5.5 now supports a number of color libraries. `ColorFamilyName` specifies the color library to use, and `ColorInkName` identifies the specific pigment. Note that the full name must be provided for `ColorInkName`.

The `Color` statement can also express a tint as a percentage of a base color. `ColorTintPercentage` specifies the percentage, and `ColorTintBaseColor` specifies the base color to use.

`ColorOverprint` is a new statement that assigns overprinting to the color. If a graphic object has no overprint statement in it, the overprint setting for that object's color is assumed.

Paragraph and Character statements

In version 5.5, the `PgfFont` and `Font` statements can now include the `FLanguage` statement to define a language for a range of text within a paragraph.

The PgfFont and Font statements include statements to describe combined fonts. For information on combined fonts, see "Combined Fonts" on page 214.

The PgfFont and Font statements include a new `FEncoding` statement to specify the encoding used for the font. The keywords for this statement are: `JISX0208.ShiftJIS, BIG5, GB2312-80.EUC, or KSC5601-1992.`

`FStretch` is a new statement to define the amount to stretch or compress a range of characters.

Text inset statements

The `TiText` and `TiTextTable` statements respectively include two new statements, `TiTxtEncoding` and `TiTxtTblEncoding`, to specify the text encoding for the source file. Both of these new statements can have one of the following keywords: `TiIsoLatin`, `TiASCII`, `TiANSI`, `TiMacASCII`, `TiJIS`, `TiShiftJIS`, `TiEUC`, `TiBig5`, `TIEUCCNS`, `TiGB`, `TiHZ`, or `TiKorean`.

Marker statements

In FrameMaker, users can define named custom markers. `MTypeName` is a new statement to specify the marker name. The `MType` statement is still written out for backward compatibility, but FrameMaker reads `MTypeName` when present.

Graphic object statements

If the `Overprint` statement is not present in a graphic object, the overprint setting for the object's color is assumed.

`ObTint` applies a tint to whatever color is assigned to the object. If the object's color already has a tint, the two tint values are added together.

Structured element definition statements

`EDAttrHidden` is a new statement in the `EDAttrDef` that specifies whether an attribute is hidden or not.

`FStretch` and `FStretchChange` are new statements added to the `FmtChangeList` to specify how much to stretch or compress the characters in an element.

Changes between versions 4 and 5

This section describes changes to MIF syntax between versions 4 and 5 of FrameMaker.

Changes to existing MIF statements

In version 5, the following MIF statements have changed or now have additional property statements.

- Paragraph statements
- Character statements
- Table statements
- Document statements
- Text frame statements
- Text flow statements
- Graphic frame statements
- Text inset and data link statements
- Structured document statements

Version 5 also introduces a new internal graphic format for imported vector graphics.

Paragraph statements

In version 5, paragraphs can span all text columns and side heads or span columns only. As a result of this change, the `PgfPlacementStyle` statement now supports the additional keyword `StraddleNormalOnly`, which indicates that the paragraph spans text columns but not side heads.

For supporting the capability to create PDF bookmarks from paragraph tags, the new `PgfAcrobatLevel` statement has been added. This statement specifies the paragraph's level in an outline of bookmarks.

For more information about the MIF syntax for paragraphs, see "Pgf statement" on page 62.

Character statements

In version 5, the `FDX`, `FDY`, and `FDW` statements, which specify the horizontal kern value, the vertical kern value, and the spread of characters, now measure in terms of the percentage of an em.

In previous versions, the `FDX` and `FDY` statements specified values in points. When reading MIF files from previous versions, FrameMaker in version 5 will convert points into the percentage of an em. Previous versions of FrameMaker generate error messages when reading `FDX` and `FDY` statements specifying percentages, since these products expect the kerning value in points.

Table statements

In version 5, tables can be aligned along the inside or outside edge (in relation to the binding of a book) of a text column or text frame. As a result of this change, the `TblAlignment` statement now supports the additional keywords `Inside` and `Outside`.

In addition, the existing `TblTitleContent` statement is now contained in the new `TblTitle` statement.

For more information about the MIF syntax for tables, see "Tbl statement" on page 79.

Document statements

In version 5, the `DAcrobatBookmarksIncludeTagNames` statement has been added under the `Document` statement to support the conversion of paragraph tags to bookmarks in Adobe Acrobat. By default, this statement is set to `No`.

Another new statement, `DGenerateAcrobatInfo`, sets print options to the required states for generating Acrobat information. By default, this statement is set to `Yes`.

For View Only documents, the default value of the `DViewOnlySelect` statement has changed from `Yes` to `UserOnly`.

For text insets, the following statement has been renamed:

MIF 4.00	MIF 5.00
<DUpdateDataLinksOnOpen *boolean*>	<DUpdateTextInsetsOnOpen *boolean*>

Document and text flow statements

In version 5, the MIF statements describing interline spacing and padding, which appeared under the `Document` statement in previous versions, have been replaced by corresponding statements under the `TextFlow` statement:

MIF 4.00	MIF 5.00
<DMaxInterLine *dimension*>	<TFMaxInterLine *dimension*>
<DMaxInterPgf *dimension*>	<TFMaxInterPgf *dimension*>

In version 5, if FrameMaker finds the `DMaxInterLine` and `DMaxInterPgf` statements in a 4.00 document, FrameMaker applies these settings to all flows in the document.

Text frame and text flow statements

Version 5 introduces *text frames*, which are composed of any number of text columns separated by a standard gap. In MIF files, text frames are described by the same statement used in previous versions for text columns, the `TextRect` statement.

In version 5, three new statements have been added under the `TextRect` statement to specify multicolumn text frames:

- `<TRNumColumns integer>`
- `<TRColumnGap dimension>`
- `<TRColumnBalance boolean>`

When reading 5.00 MIF files, previous versions of FrameMaker will remove these statements and assume that the text frame is actually a single text column.

When reading MIF files from previous versions, FrameMaker in version 5 will convert multiple text columns on a page into a single, multicolumn text frame. To represent each text column as a separate text frame, include the MIF statement `<TRNumColumns 1>` in the description of each `TextRect` statement.

Side head layout information has been transferred from the `TextFlow` statement to the `TextRect` statement. The following statements, which appeared under the `TextFlow` statement in previous versions, are replaced by corresponding statements under the `TextRect` statement in 5.00:

MIF 4.00	MIF 5.00
`<TFSideheadWidth dimension>`	`<TRSideheadWidth dimension>`
`<TFSideheadGap dimension>`	`<TRSideheadGap dimension>`
`<TFSideheadPlacement keyword>`	`<TRSideheadPlacement keyword>`

If FrameMaker in version 5 finds the `TextFlow` MIF statements for side heads, FrameMaker will convert these statements to the equivalent statements under the `TextRect` statement.

If these types of statements are found under both the `TextRect` statement and the `TextFlow` statement, the statements under the `TextRect` statement will be used.

Note that the existence of side heads in a text flow is still specified by the `TFSideheads` statement, which is under the `TextFlow` statement.

For more information about the MIF syntax for text frames, see "TextRect statement" on page 129. For more information about the MIF syntax for text flows, see "Text flows" on page 130.

Graphic frame statements

In version 5, graphic frames can be anchored inside or outside text frames. Graphic frames can also be aligned along the inside or outside edge of a text frame (in relation to the binding of a book). Finally, graphic frames can be anchored outside the entire text frame or one column in the text frame.

As a result, the following changes to 4.00 MIF have been made:

- The `FrameType` statement now supports the additional keywords `Inside`, `Outside`, and `RunIntoParagraph`.
- The `AnchorAlign` statement now supports the additional keywords `Inside` and `Outside`.
- Version 5 introduces the new `AnchorBeside` statement to indicate whether the graphic frame is anchored outside the entire text frame (`TextFrame`) or outside one column in the text frame (`Column`).
- When editing FrameMaker document files from previous versions, FrameMaker assumes that this statement has the value `<AnchorBeside Column>`.

For more information about the MIF syntax for graphic frames, see "Frame statement" on page 117.

Text inset and data link statements

In previous versions, Macintosh versions of FrameMaker allowed you to import text by reference with the Publish and Subscribe mechanism. The MIF `DataLink` statement described text that was published or subscribed.

In version 5, the capability to import text by reference, which creates a text inset, is available on all platforms. As a result of this new feature, the new `TextInset` statement replaces the `DataLink` statements for subscribers.

Note that the `DataLink` statements for publishers are still used.

The following table lists the old `DataLink` statements and the new `TextInset` statements that replace them.

MIF 4.00	MIF 5.00
`<DataLink...>`	`<TextInset...>`
`<DLSource` *pathname*`>`	`<TiSrcFile` *pathname*`>`
`<DLParentFormats Yes>`	`<TiFormatting TiEnclosing>`
`<DLParentFormats No>`	`<TiFormatting TiSource>`
`<OneLinePerRec` *boolean*`>`	`<EOLisEOP` *boolean*`>`
`<MacEdition` *integer*`>`	`<TiMacEditionId` *integer*`>`
`<DataLinkEnd>`	`<TextInsetEnd>`

If you open a 5.00 MIF file with text insets in a version 4 FrameMaker product, the older version of the product will strip out the text inset MIF statements. The text inset becomes plain text that cannot be updated.

For more information about the MIF syntax for text insets, see "Text insets (text imported by reference)" on page 138. For information about the MIF syntax for publishers, see "If the `TiTblNumHdrRows` substatement is not set to `0`, the table has header rows. If the `TiTblHeadersEmpty` substatement is set to `No`, these rows are filled with imported text." on page 144.

Structured document statements

In version 5, FrameMaker does not support statements for structured documents, such as `ElementDefCatalog` and `DElementBordersOn`. FrameMaker strips these statements when reading in a MIF file. When writing out a MIF file, FrameMaker does not write these statements.

FrameVector graphic format

The internal graphic format FrameVector is supported for imported vector graphics. The specifications for this facet are described in , "FrameVector Facet Format."

Changes between versions 3 and 4

This section describes the changes to MIF syntax between versions 3 and 4 of FrameMaker.

4.00 top-level MIF statements

The following table lists top-level statements introduced between versions 3 and 4 of FrameMaker.

New statement	Action in earlier versions
`<ColorCatalog...>`	All custom colors revert to Cyan

New statement	Action in earlier versions
`<Views…>`	Ignored

Changes to 3.00 MIF statements

This section describes the statements that have changed or that have introduced additional property statements between versions 3 and 4 of FrameMaker. MIF statements that have changed include:

- Color statements
- Math statements
- Character format statements
- Object statements
- Page statements

Color statements

The following table lists the changes for color property statements.

MIF 3.00	MIF 4.00
`<FSeparation integer>`	`<FColor string>`
`<CSeparation integer>`	`<CColor string>`
`<RulingSeparation integer>`	`<RulingColor string>`
`<Separation integer>`	`<ObColor string>`
`<TblHFSeparation integer>`	`<TblHFColor string>`
`<TblBodySeparation integer>`	`<TblBodyColor string>`
`<TblXSeparation integer>`	`<TblXColor string>`
`<CellSeparation integer>`	`<CellColor string>`
`<DChBarSeparation integer>`	`<DChBarColor string>`

`Separation` values refer to the reserved, default colors that appear in the Color pop-up menu in the FrameMaker Tools palette.

This value	Corresponds to this color
`<Separation 0>`	Black
`<Separation 1>`	White
`<Separation 2>`	Red
`<Separation 3>`	Green
`<Separation 4>`	Blue
`<Separation 5>`	Cyan
`<Separation 6>`	Magenta
`<Separation 7>`	Yellow
`<Separation 8>`	Dark Grey

This value	Corresponds to this color
`<Separation 9>`	Pale Green
`<Separation 10>`	Forest Green
`<Separation 11>`	Royal Blue
`<Separation 12>`	Mauve
`<Separation 13>`	Light Salmon
`<Separation 14>`	Olive
`<Separation 15>`	Salmon

Version 4 and later versions of FrameMaker read separation statements and convert them to the equivalent color statements. FrameMaker writes both color statements and separation statements for backward compatibility. For the reserved default colors, FrameMaker writes the equivalent separation value. For custom colors, FrameMaker writes the separation value `5` (Cyan) so that you can easily find and change custom colors.

If your application creates files that will be read by both older (before version 4) and newer (after version 4) FrameMaker product versions, include both color and separation statements in the MIF files; otherwise, use only the color statements.

Math statements

The following table lists the changes for math statements.

MIF 3.00	MIF 4.00
`DMathItalicFunctionName`	`DMathFunctions`
`DMathItalicOtherText`	`DMathNumbers, DMathStrings, DMathVariables`

In addition, the `diacritical` expression defines new diacritical marks (see "Using char and diacritical for diacritical marks" on page 200). The `diacritical` expression is not backward compatible.

Character format statements

The following table lists the changes in `Font` and `PgfFont` statements.

MIF 3.00	MIF 4.00
`<FUnderline` *`boolean`*`>`	`<FUnderlining FSingle>`
`<FDoubleUnderline` *`boolean`*`>`	`<FUnderlining FDouble>`
`<FNumericUnderline` *`boolean`*`>`	`<FUnderlining FNumeric>`
`<FSupScript` *`boolean`*`>`	`<FPosition FSuperscript>`
`<FSubScript` *`boolean`*`>`	`<FPosition FSubscript>`

If your application only reads or writes files for version 4 or later versions of FrameMaker, use only the 4.00 statements. If your application reads or writes files for version 3 or previous versions of FrameMaker, use only the 3.00 statements. Do not use both statements.

The MIF interpreter always reads the MIF 3.00 statements. It writes both 3.00 and 4.00 statements for backward compatibility.

Object statements

The following table lists the changes in graphic object statements (see "Graphic objects and graphic frames" on page 111).

MIF 3.00	MIF 4.00
`<Angle 0|90|180|270 >`	`<Angle degrees>`
`<BRect>`	`<ShapeRect>`

Text lines, text frames, imported graphics, table cells, and equations that are rotated at an angle of 90, 180, or 270 degrees retain rotation in earlier versions. If these objects are rotated at any other angle, they are rotated back to 0 degrees in the earlier version. All other objects are rotated back to 0 degrees.

FrameMaker writes both `BRect` and `ShapeRect` values for backward compatibility. For text lines, text frames, imported graphics, table cells, and equations that are rotated at an angle of 90, 180, or 270 degrees, the `BRect` value is the position and size of the object *after* rotation. For any object rotated at any other angle, the `BRect` value is the position and size of the object *before* rotation, which is the same as the `ShapeRect` value.

Device-independent pathnames

The following codes for pathname components in a device-independent pathname are obsolete and are ignored by the MIF interpreter.

Code	Meaning
A	Apollo-dependent pathname
D	DOS-dependent pathname
M	Macintosh-dependent pathname
U	UNIX-dependent pathname

For information about valid codes, see "Device-independent pathnames" on page 7.

Document statements

The following changes have been made to `Document` statements.

MIF 3.00	MIF 4.00
`<DCollateSeparations boolean>`	`<DNoPrintSepColor> and <DPrintProcessColor>`

In addition, the `Document` statement has a number of new property statements that set options for View Only documents (see page 93), set options for structured documents, and define custom math operators (see page 190).

Page statement

The following change has been made to the `Page` statement.

MIF 3.00	MIF 4.00
`<PageOrientation keyword>`	`<PageAngle> and <DPageSize>`

A page's size and orientation (landscape or portrait) is determined by the `PageAngle` statement and the `Document` substatement `DPageSize`. FrameMaker writes the `PageOrientation` statement for backward compatibility. MIF generators should use the `PageAngle` statement instead of `PageOrientation`.

When the MIF interpreter reads a `Page` statement that includes both a `PageAngle` and a `PageOrientation` statement, it ignores the `PageOrientation` statement. When the interpreter reads a `Page` statement that contains a `PageOrientation` statement but no `PageAngle` statement, it determines the page's angle from the `PageOrientation` statement. If the page orientation matches the orientation determined by the `DPageSize` statement, the page's angle is 0 degrees; otherwise, the page's angle is 90 degrees. A page that has neither a `PageAngle` nor a `PageOrientation` statement has an angle of 0 degrees.

Chapter 11: Facet Formats for Graphics

When you copy a graphic into an Adobe® FrameMaker® document, the FrameMaker document stores the graphic data in one or more *facets*. Each facet contains data in a specific graphic format. FrameMaker uses facets to display and print graphics.

In UNIX versions of FrameMaker, you can associate a graphic application with FrameMaker through the FrameMaker API or through the FrameServer interface. You can set this up so that the graphics created and modified in the graphic application can be imported directly into a FrameMaker document. The graphic application becomes a *graphic inset editor*. Graphic inset editors write graphic data to *graphic insets*, which can be read by FrameMaker.

For more information on setting up graphic inset editors, see the *FDK Programmer's Guide* and the online manual, *Using FrameServer with Applications and Insets*. Both manuals are provided with the UNIX version of the Frame Developer's Kit.

The first part of this appendix describes the general format for a facet in a MIF file. The second part of this appendix explains the graphic inset format.

Note: *If you are using the API to implement the graphic inset editor, the syntax described in this appendix applies only to external graphic insets. For information on specifying facet names, data types, and data for internal graphic insets, see the FDK Programmer's Guide.*

Facets for imported graphics

A graphic imported by copying into a FrameMaker document contains one or more *facets*. Each facet describes the imported graphic in a specific graphic format. All imported graphics copied into a document contain one or more facets used to display and print the file.

FrameMaker might not use the same facet for displaying and printing a graphic.

When printing an imported graphic, FrameMaker selects one of the following facets (in order of preference):

- EPSI (Encapsulated PostScript)
- Native platform facet (QuickDraw PICT, WMF)
- FrameVector
- TIFF
- FrameImage and other bitmap facets

When displaying an imported graphic, FrameMaker selects one of the following facets (in order of preference):

- Native platform facet (QuickDraw PICT, WMF)
- FrameVector
- FrameImage
- TIFF
- Other bitmap facets

All versions of FrameMaker recognize EPSI (with DCS Cyan, DCS Magenta, DCS Yellow, and DCS Black for color separations), TIFF, FrameImage, and FrameVector facets. Windows versions of FrameMaker recognize WMF and OLE facets.

If the graphic data does not have a corresponding facet supported by FrameMaker for displaying or printing, FrameMaker can use filters to convert the graphic data into one of two internal facets: FrameImage (for bitmap data) and FrameVector (for vector data). For example, FrameMaker does not have a facet for HPGL, so HPGL data is converted into a FrameVector facet.

In Windows versions of FrameMaker, users can choose to automatically save a cross-platform facet of an imported graphic. If a cross-platform facet does not already exist, FrameMaker generates a FrameImage facet for the imported graphic.

Basic facet format

A facet consists of a facet name, a data type, and a series of lines containing facet data. For example:

```
=EPSI
&%v
&%!PS-Adobe-2.0 EPSF-2.0\n
```

Facet name

The first line of a facet identifies the facet by name. The facet name line has the following format:

`=`*`facet_name`*

The facet name can be one of the standard display and print facets or an application-specific name registered with FrameMaker. (For information about registering your application-specific facets, see the *FDK Platform Guide* for your platform, which is included with the Frame Developer's Kit.)

Data type

The second line provides the data type of the facet: unsigned bytes (`&%v`), integer (`&%i`), or metric (`&%m`).

If the facet data is binary (such as FrameImage and FrameVector data) or if it contains ASCII characters (such as EPSI data, as shown in the preceding example), the facet uses the unsigned bytes data type (`&%v`).

For example, the following line is the second line in a facet that contains data represented as unsigned bytes:

```
&%v
```

Facet data

The remaining lines contain the facet data. Each line begins with an ampersand (&).

The end of the data for a facet is marked by the beginning of a new facet. Thus, a line with a new facet name signals the end of the previous facet data.

The end of the last facet in the graphic inset is marked by the following line:

```
=EndInset
```

Unsigned bytes

If the facet data contains a backslash character, another backslash precedes it as an escape character. For example, if the data contains the string `x\yz`, the facet contains `x\\yz`.

Within the facet data, nonprintable ASCII characters or non-ASCII bytes (greater than `7f`) are represented in hexadecimal.

Any section of data represented in hexadecimal is preceded and followed by the characters `\x`. For example, the following FrameImage facet contains data represented in hexadecimal, which is enclosed between two sets of `\x` characters:

```
=FrameImage
&%v
&\x
&59a66a95
&00000040
...
&0000FC0001FC0000
&\x
=EndInset
```

Integer data

The integer data type stores integer values in a facet. For example, the `fmbitmap` program stores the dimensions of the graphic, the x-coordinate of the hot spot, and the y-coordinate of the hot spot as integer data in a facet:

```
=Data.facet
&%i
&64
&64
&-1
&-1
```

Metric data

Metric data describes a graphic in terms of units of measurement. The following table shows the abbreviations used to denote units within a facet.

Units	Abbreviation
Centimeters	`cm`
Ciceros	`cicero`, `cc`
Didots	`dd`
Inches	`in`, `"`
Millimeters	`mm`
Picas	`pica`, `pi`, `pc`
Points	`point`, `pt`
pixels	px

Graphic insets (UNIX versions)

A graphic inset contains graphic data that can be written by a graphic application and used by FrameMaker to display and print an imported graphic. A graphic inset can also specify a live link, which associates an imported graphic in a FrameMaker document with the graphic application used to edit the graphic. A live link can be set up through FrameServer functions or through an FDK client.

When a live link is established between an imported graphic and a graphic application, users can edit the graphic in a graphic application and directly import the graphic into a FrameMaker document. For more information on live links, see the *FDK Programmer's Guide*, which is provided with the FDK, or the online manual, *Using FrameServer with Applications and Insets*, which is provided with the UNIX version of the FDK.

To set up a live link between a graphic application and a FrameMaker document, you need to add functions to your application to write out graphic data as a graphic inset.

A graphic inset consists of an `ImportObject` statement that contains one or more facets for display and print. If your application requires additional information not supported by the display and print facet, the graphic inset also needs one or more application-specific facets to store this additional information.

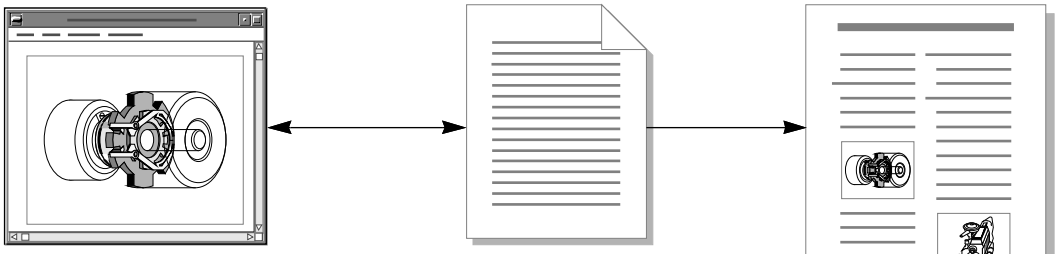
The two types of graphic insets are *internal graphic insets* and *external graphic inset files*. Each type results in a slightly different type of integration between FrameMaker and your application. You can choose the type of graphic inset that your application supports. In most cases, one format is adequate, but you might want to give users more than one option. Both types require a display and print facet.

External graphic insets

An external graphic inset file remains independent of the FrameMaker document. The FrameMaker document contains only a pathname for the graphic inset file. Because the graphic inset data is not contained in the FrameMaker document, users can access the graphic inset data from FrameMaker, from your application, or from another application.

To edit an external graphic inset from FrameMaker, users must open FrameMaker document, select the graphic inset, and choose the Graphic Inset command from the Special menu. FrameMaker passes the external graphic inset filename to your application and instructs your application to edit the graphic inset. When users finish editing a graphic inset, they issue your application's command for pasting a graphic inset to FrameMaker, and FrameMaker immediately updates the graphic inset file.

If users edit the graphic inset from another application, FrameMaker displays the updated graphic inset the next time the document is opened. Note that if the graphic inset file is moved or deleted, FrameMaker will be unable to display the data and will inform the user that the graphic inset is missing.



External graphic insets are best suited to situations in which users are documenting projects in progress or in which the document's graphics are updated by external sources (for example, by a database).

An external graphic inset file contains a `MIFFile` statement and an `ImportObject` statement. The `ImportObject` statement lists the graphic inset file's pathname, the name of the inset editor that created it, and all of its facets.

An external graphic inset file has the following format:

```
<MIFFile 2015>
<ImportObject
    <ImportObEditor inset_editor_name>
    <ImportObFileDI device_independent_pathname>
=facet_name
&data_type
&facet_data
...
=facet_name
&data_type
&facet_data
...
=EndInset
>
```

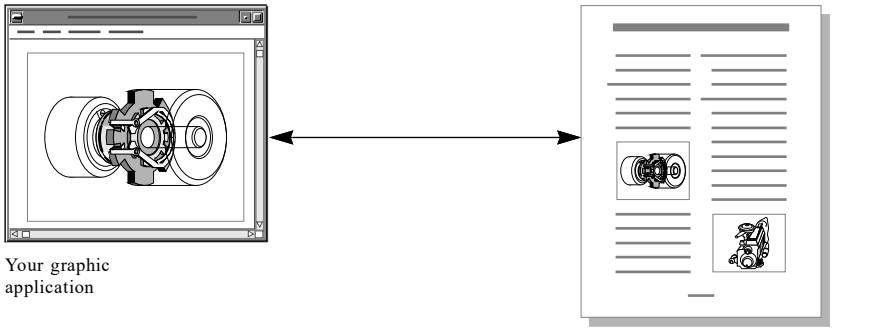
A MIF `ImportObEditor` statement names the main editor for application-specific facets in the graphic inset file.

A MIF `ImportObFileDI` statement specifies the device-independent pathname for the graphic inset file. For more information on device-independent pathnames, see the section "Device-independent pathnames" on page 7.

Internal graphic insets

An internal graphic inset is entirely contained within FrameMaker document file. Once the link is established, the graphic inset data exists only in FrameMaker document.

Users can access the graphic only through FrameMaker. To edit an internal graphic inset, users must open FrameMaker document, select the graphic inset, and choose the Graphic Inset command from the Special menu. FrameMaker writes the graphic inset to a temporary file and instructs your application to edit it.



Your graphic application

FrameMaker document with internal graphic inset

Internal graphic insets are best suited for environments in which portability of FrameMaker document across different types of systems is most important.

When FrameMaker creates temporary files for internal graphic insets, the temporary files have the following format:

```
<MIFFile 2015>
<ImportObject
    <ImportObEditor inset_editor_name>
    <ImportObFile `2.0 internal inset'>
=facet_name
&data_type
&facet_data
...
=facet_name
&data_type
```

```
&facet_data
...
=EndInset
>
Because the graphic inset is stored in FrameMaker document, the file does not have an
ImportObFileDI statement.
```

The `ImportObFile` statement identifies the file as a FrameMaker version 2.0 internal graphic inset file for compatibility with earlier versions of FrameMaker. If you do not plan to use the graphic insets generated by your application with earlier versions of FrameMaker, you can omit this statement.

Application-specific facets

Application-specific facets can be in any format your application understands, and a graphic inset file can contain as many application-specific facets as you want.

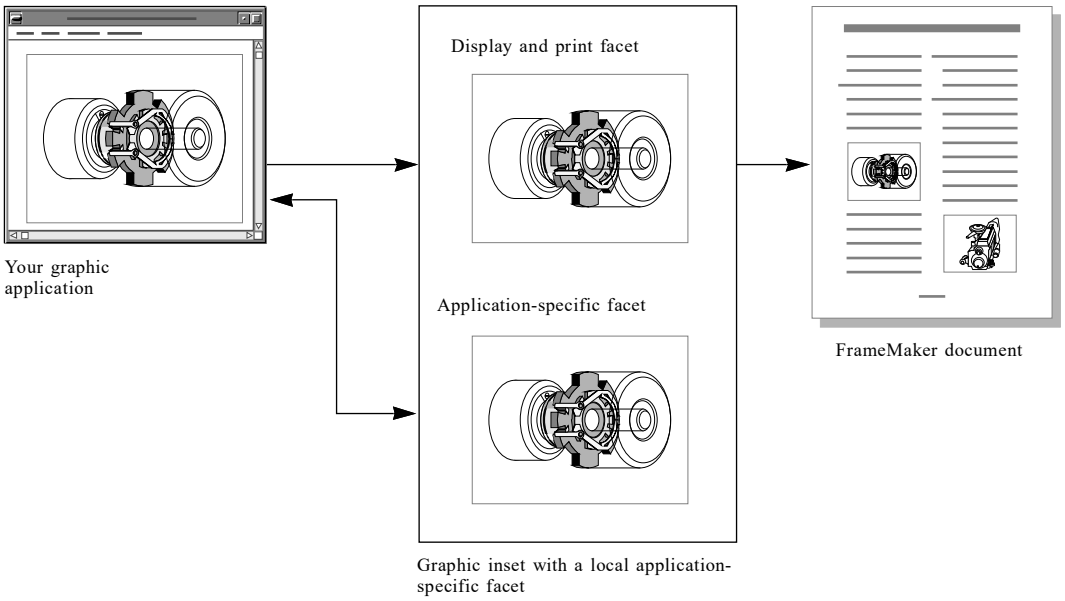
When selecting application-specific facets for your graphic inset file, you might want to include an industry-standard facet (for example, EDIF for EDA applications) so that you can use the graphic inset file to share data with applications other than FrameMaker.

Application-specific facets can be contained entirely within the graphic inset file (a *local facet*), or the graphic inset file can contain a reference to an external data file or database (a *remote facet*).

Local application-specific facets

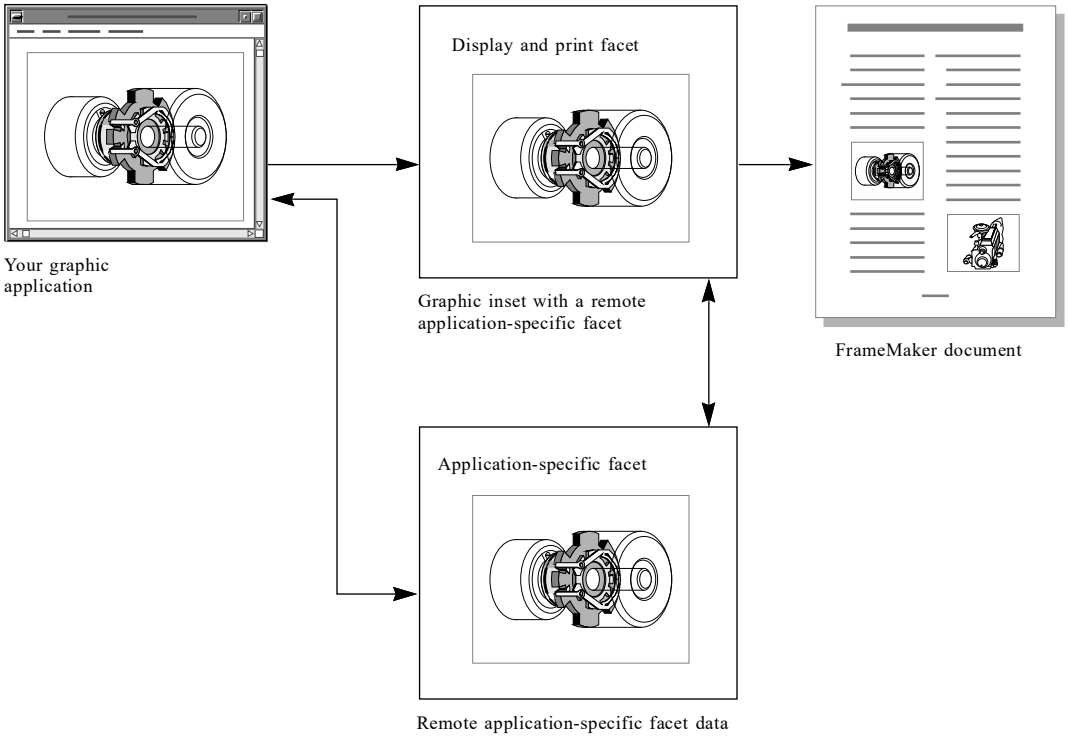
A local application-specific facet is contained in the graphic inset file. The formats for external and internal graphic insets (described in the sections "External graphic insets" on page 270 and "Internal graphic insets" on page 271) apply to local application-specific facets.

The following illustration shows the relationship between your application, FrameMaker document, and a graphic inset file with a local application-specific facet.



Remote application-specific facets

A remote application-specific facet contains the pathname or database key for an existing data file or database. Since application-specific data is normally duplicated in a separate application file, remote facets conserve file space. Because the application-specific facet contains only a pathname, remote facets are easier to implement.



Note: *Display and print facets must be contained in the graphic inset file. They cannot be remote facets.*

To write a remote facet, your graphic application must write an application data file and store its data type and pathname in the graphic inset file. A remote application-specific facet has the following format:

```
=facet_name
&facet_type
&path_for_facet_file
=EndInset
```

For example, the following lines describe the remote facet described in the application data file `/diagrams/BlockDiagram`:

```
=application_name.facet
&%v
&/diagrams/BlockDiagram
=EndInset
```

Example of graphic inset file

The following example is the external graphic inset file generated by the `fmbitmap` program, which is shipped with the UNIX version of the FDK.

The graphic inset file is named `/tmp/default.fi`. The application-specific facet for this graphic inset (the file generated by the `fmbitmap` program) is stored in a remote facet in the file `/tmp/default.`

Note that although the `fmbitmap` program writes out the `ImportObFile` statement, this statement is obsolete and is only used with older versions of FrameMaker. When defining a function to write a graphic inset file, use the `ImportObFileDI` statement and specify a device-independent pathname. For more information on device-independent pathnames, see "Device-independent pathnames" on page 7.

```
<MIFFile 2015>                    # Generated by fmbitmap
    <ImportObject
        <ImportObFile /tmp/default.fi>
        <ImportObEditor fmbitmap>
=BitmapFile.facet
&%v
&/tmp/default
=Data.facet
&%i
&64
&64
&-1
&-1
=FrameImage
&%v
&\x
& ...
&\x
=EndInset
    >
```

To see more examples of the graphic inset format, you can import a graphic into a FrameMaker document (import by copying) and save the FrameMaker document as a MIF file.

General rules for reading and writing facets

To write a facet, you need to modify the existing function in your application for writing data. The function must write the facet name and data type lines and insert an ampersand at the beginning of each line of facet data. If necessary, convert data lines to the appropriate facet data format. Unsigned bytes should follow the conventions described in "Unsigned bytes" on page 268, and metric data should follow the conventions described in "Metric data" on page 269.

When writing the facet data, your application can use as many lines as necessary. Each line should be short enough to read with a text editor, in case you need to debug the graphic inset file. There are no counts, offsets, or facet size limits.

Facet data in hexadecimal must contain valid hexadecimal digits only (0-9, A-F) and cannot contain backslash (\) characters. When you write a facet containing hexadecimal data, do not write newline characters (`\r` or `\n`) at the end of the lines.

Graphic insets cannot contain any blank lines within or between facets.

When reading a graphic inset, your application need only scan for facet name lines and then read the appropriate facets. Since facets begin and end with the `=facet_name` token, your program should read facet data until it encounters an equal sign in column 1.

If your application encounters the characters `\x` when reading facet data, it should process the subsequent data as hexadecimal until it encounters another `\x`. If your facet contains a mix of ASCII characters and hexadecimal data, it might be simpler for you to represent the ASCII characters as character codes in hexadecimal. For example, the FrameVector format represents strings (such as `black`) as character codes in hexadecimal (such as `62 6c 61 63 6b`).

Chapter 12: EPSI Facet Format

EPSI is an interchange standard developed by Adobe Systems Incorporated. You can obtain a complete specification of the EPSI format from Adobe Systems Incorporated.

Imported graphics can contain graphic data in EPSI format. This data is called the EPSI facet of the graphic. Adobe® FrameMaker® can use this facet to display and print the graphic. For more information about facets, see , "Facet Formats for Graphics."

In a MIF file, the EPSI facet is contained in the `ImportObject` statement. For more information about the statement, see "ImportObject statement" on page 120.

Specification of an EPSI facet

An EPSI facet begins with the following facet name and data type lines:

```
=EPSI
&%v
```

Each line of EPSI facet data ends with `\n`.

When FrameMaker imports a graphic inset with an EPSI facet, FrameMaker uses the EPSI bounding box to determine the graphic inset's size. If the bounding box does not fit on the page, FrameMaker halves its dimensions until it fits.

Example of an EPSI facet

The following rectangle is an imported graphic:

The following MIF statements describe the imported graphic. The graphic data that specifies the rectangle is an EPSI facet.

```
<ImportObject
    <BRect 0 0 0.25" 0.25">
    <Pen 15> <Fill 15>
    <ImportObFile `2.0 internal inset'>
=EPSI
&%v
&%!PS-Adobe-2.0 EPSF-2.0\n
&%%BoundingBox: 0 0 18 18\n
&%%Pages: 0\n
&%%Creator: contr2\n
&%%CreationDate: Tue Apr 25 16:09:56 1989\n
&%%EndComments\n
&%%BeginPreview: 18 18 1 18\n
&%FFFFC0\n
```

```
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%FFFFC0\n
&%%EndPreview\n
&%%EndProlog\n
&%%Page: "one" 1\n
&0 0 moveto 18 0 rlineto 0 18 rlineto -18 0 rlineto closepath 0 setgray\n
&fill\n
&%%Trailer\n
=EndInset
>                   # End ImportObject
```

Chapter 13: FrameImage Facet Format

FrameImage is a format for bitmap graphics that is recognized by Adobe® FrameMaker® on all platforms. The specification of the FrameImage format is documented in this appendix.

Imported graphics can contain graphic data in FrameImage format. This data is called the *FrameImage facet* of the graphic. FrameMaker can use this facet to display and print the graphic. For more information about facets, see , "Facet Formats for Graphics."

In a MIF file, the FrameImage facet is contained in the `ImportObject` statement. For more information about the statement, see "ImportObject statement" on page 120.

Specification of a FrameImage facet

A FrameImage facet begins with the following facet name and data type lines:

```
=FrameImage
&%v
```

When importing a graphic with a FrameImage display and print facet, FrameMaker prompts the user to specify the graphic inset's print resolution in the Imported Graphic Scaling dialog box. The print resolution determines the size of the imported graphic.

Specification of FrameImage data

A description of a graphic in FrameImage format consists of three parts:

- A header, which describes the dimensions and other characteristics of the graphic
- An optional color map, included only if the graphic uses colors
- Data describing the bitmap of the imported graphic

The description is written as integer values in hexadecimal format. Each line is preceded by an ampersand (`&`). The data section begins with the `%v` characters, which indicate that the FrameImage data is represented as unsigned bytes. The beginning and end of the data are bracketed by the symbol `\x`, which indicates that the data is in hexadecimal format.

Header

The header describes properties of the imported graphic. These properties are described by eight 32-bit integer values, such as the values shown in the following example:

```
&59a66a95
&00000040
&00000040
&00000001
&00000000
&00000001
&00000000
&00000000
```

Each value identifies a property of the imported graphic:

- The first value is always the constant value `0x59a66a95`.
- The second value is the width of the graphic in pixels. In the preceding example, the graphic is 64 pixels wide (converting the hexadecimal value `0x00000040` to the decimal value `64`).
- The third value is the height of the graphic in pixels. In the example, the graphic is 64 pixels high (converting the hexadecimal value `0x00000040` to the decimal value `64`).
- The fourth value is the number of bits used to describe a single pixel. This value is sometimes referred to as the depth of the graphic. For black and white graphics, only one bit is used to describe a single pixel. For color images, eight bits are used to describe a single pixel. In the example, the value `0x00000001` indicates that the graphic is in black and white.
- The fifth value is not currently used and is set to `0x00000000` by default.
- The sixth value specifies whether or not the data is encoded. If the data is encoded, this value is set to `0x00000002`. If the data is not encoded (that is, if the data is in uncompressed format), this value is set to `0x00000001`. In the example, the data is uncompressed.
- The seventh value identifies the type of color map used by the graphic. If the graphic is in black and white, no color map is used, and this value is set to `0x00000000`. If the graphic is in color, an RGB color map is used, and this value is set to `0x00000001` or `0x00000002`. In the example, because the graphic is in black and white, the value is set to `0x00000000`.
- The eighth value is the length of the color map in bytes. If the graphic is in black and white, no color map is used, and this value is set to `0x00000000`. If the graphic is in color, a color map with 256 colors is used (described by 768 bytes of information), and this value is set to `0x00000300` (the hexadecimal representation of the number 768). In the example, because the graphic is in black and white, the value `0x00000000` is used.

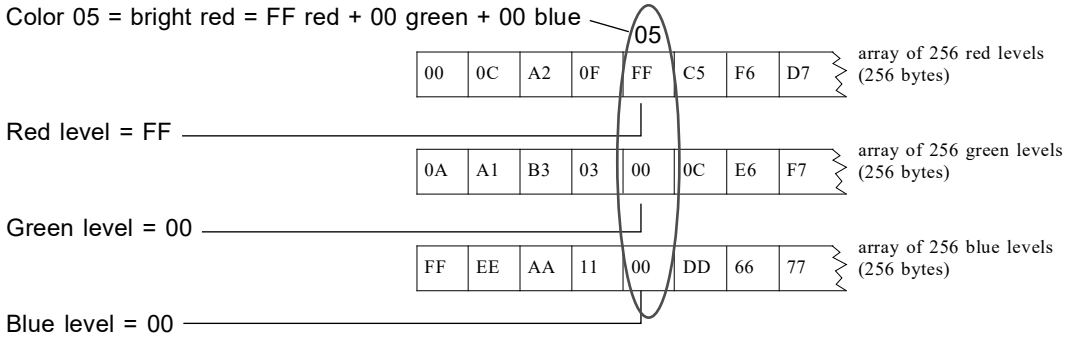
The FrameImage format is similar to the Sun rasterfile format for bitmap images. The following section of code is part of the `/usr/include/rasterfile.h` header file, which describes the Sun rasterfile format:

```
...
struct rasterfile {
    IntT ras_magic; /* magic number */
    IntT ras_width; /* width (pixels) of image */
    IntT ras_height; /* height (pixels) of image */
    IntT ras_depth; /* depth (1, 8, or 24 bits) of pixel */
    IntT ras_length; /* length (bytes) of image */
    IntT ras_type; /* type of file; see RT_* below */
    IntT ras_maptype; /* type of colormap; see RMT_* below */
    IntT ras_maplength; /* length (bytes) of following map */
/* color map follows for ras_maplength bytes, followed by image */
};
                #define RAS_MAGIC 0x59a66a95
/* Sun supported ras_type's */
...
                #define RT_STANDARD 1 /* Raw image in 68000 byte order */
                #define RT_BYTE_ENCODED 2 /* Run-length compression of bytes */
...
                /* Sun registered ras_maptype's */
                #define RMT_RAW 2
                /* Sun supported ras_maptype's */
                #define RMT_NONE 0 /* ras_maplength is expected to be 0 */
                #define RMT_EQUAL_RGB 1 /* red[ras_maplength/3],green[],blue[] */
...
```

For more information, see the `/usr/include/rasterfile.h` header file and the Sun man page on `rasterfile`.

Color map

The optional color map defines colors used for the imported graphic. It consists of 256 bytes of red, followed by 256 bytes of green, followed by 256 bytes of blue. Each byte contains an intensity value for a color. FF is the maximum intensity and 00 is the minimum (none).



The color map defines 256 colors. Each color contains a red, green, and blue level of intensity. The values of the first red byte, first green byte, and first blue byte define the first color in the map; the values of the second red, green, and blue bytes define the second color, and so forth.

For example, the data value 05 represents the color defined by the level of red stored in the fifth byte of red, the level of green stored in the fifth byte of green, and the level of blue stored in the fifth byte of blue. If the fifth byte of red contains FF (the maximum red intensity) and the fifth bytes of green and blue are both 00, then 05 would represent bright red.

Data describing the graphic

The data type can be either byte encoded or standard. Each type uses different data formats.

Byte-encoded data

If `ras_type` is `RT_BYTE_ENCODED` (if the sixth value in the header is `0x00000002`), the data is a run-length encoded pixel matrix. The byte value 80 hexadecimal (decimal 128) is used as a separator for encoding several bytes of the same color. The encoding scheme uses the following format:

80 *nn pp*

where *nn*+1 is the number of times to repeat the data byte (*pp*).

For example, the following values represent seven data bytes of the hex value 55:

80 06 55

A single pixel value of 80 must be encoded as 80 00 in the data. If the value 80 occurs sequentially, use the format:

80 *nn* 80

where *nn*+1 is the number of times 80 occurs.

Standard data

If `ras_type` is `RT_STANDARD` (if the sixth value in the header is `0x00000001`), the data contains uncompressed hex data corresponding to the graphic. Each byte is eight pixels for a monochrome graphic or one pixel for color. Each scanline of data must be padded to a word (16 bit) boundary.

Differences between monochrome and color

There are two types of FrameImage files: monochrome and pseudocolor.

Monochrome images

A monochrome graphic has the following header properties:

Property	Value
ras_depth	1
ras_maptype	RMT_NONE
ras_maplength	0

An example of the header for a monochrome graphic is shown below:

```
&59a66a95
&00000040
&00000040
&00000001
&00000000
&00000001
&00000000
&00000000
```

A monochrome graphic has no color map. Each data byte represents eight pixels, and the most significant bit is the leftmost pixel.

Graphic data bytes are hex values that represent bit patterns of black and white. For example, hex 55 represents binary 01010101, which produces a gray shade; hex FF represents binary 11111111, which produces black; and hex 00 represents binary 00000000, which produces white.

Pseudocolor and gray images

A pseudocolor or gray graphic has the following header properties:

Property	Value
ras_depth	8
ras_maptype	RMT_EQUAL_RGB or RMT_RAW
ras_maplength	300

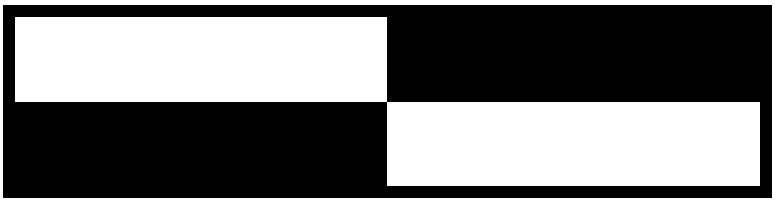
An example of the header for a color graphic is shown below:

```
&59a66a95
&00000040
&00000040
&00000008
&00000000
&00000001
&00000002
&00000300
```

Each graphic data byte represents one pixel of a particular color. The value of a data byte is an index to a color stored in the color map. (See "Color map" on page 279.)

Sample unencoded FrameImage facet

The sample FrameImage facet in this section describes the following illustration. Note that no color map is included in the description, because the graphic is in black and white.



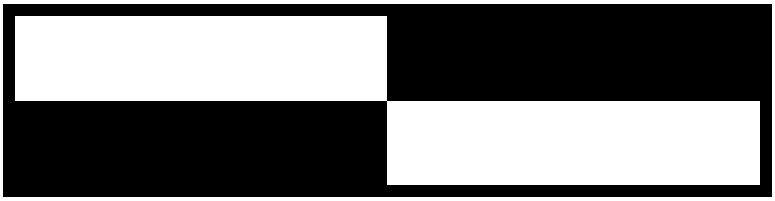
Header

Graphic data

```
...
=FrameImage
&%v
&\x
&59a66a95
&00000040
&00000010
&00000001
&00000000
&00000001
&00000000
&00000000
&FFFFFFFFFFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&80000000FFFFFFFF
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFF00000001
&FFFFFFFFFFFFFFFF
&\x
=EndInset
...
```

Sample encoded FrameImage facet

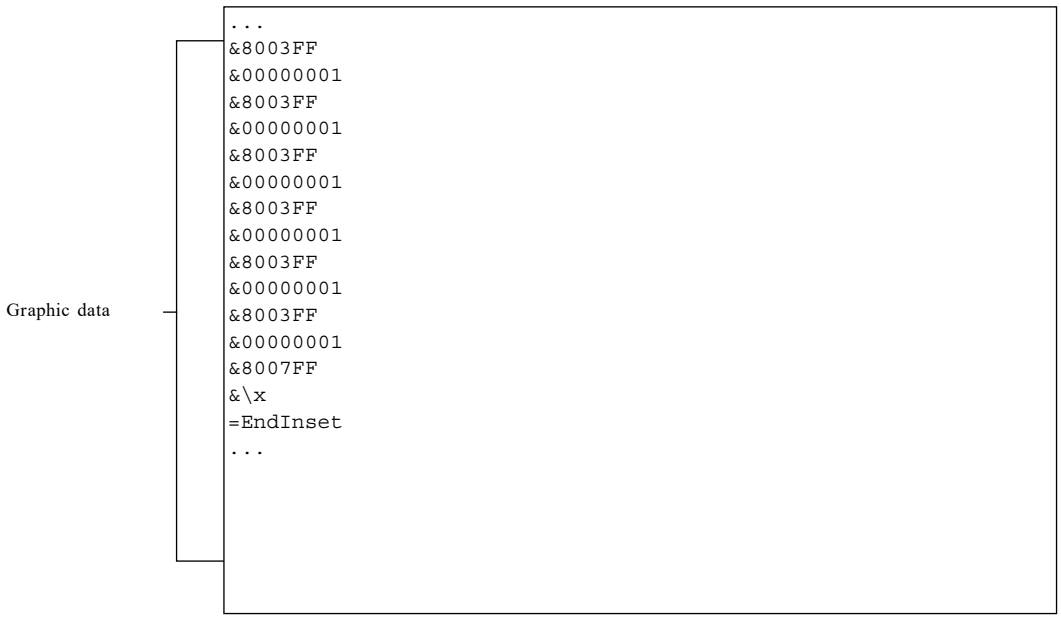
The sample FrameImage facet in this section describes the same illustration. Note that no color map is included in the description, because the graphic is in black and white. Unlike the previous file, this graphic file is in encoded format.



Header

Graphic data

```
...
=FrameImage
&%v
&\x
&59A66A95
&00000040
&00000010
&00000001
&00000000
&00000002
&00000000
&00000000
&8007FF
&8000000000
&8003FF
&8000000000
&8003FF
&8000000000
&8003FF
&8000000000
&8003FF
&8000000000
&8003FF
&8000000000
&8007FF
&00000001
...
```


Graphic data
...
&8003FF
&00000001
&8003FF
&00000001
&8003FF
&00000001
&8003FF
&00000001
&8003FF
&00000001
&8003FF
&00000001
&8007FF
&\x
=EndInset
...

Chapter 14: FrameVector Facet Format

FrameVector is a format for vector graphics that is recognized by Adobe® FrameMaker® on all platforms. The specification of the FrameVector format is documented in this appendix.

Imported graphics can contain graphic data in FrameVector format. This data is called the *FrameVector facet* of the graphic. FrameMaker can use this facet to display and print the graphic. For more information about facets, see , "Facet Formats for Graphics."

In a MIF file, the FrameVector facet is contained in the `ImportObject` statement. For more information about the statement, see "ImportObject statement" on page 120.

Specification of a FrameVector facet

A FrameVector facet begins with the following facet name, facet data type, and version number lines:

```
=FrameVector
&%v
&<MakerVectorXXX>
```

In the version number line, XXX is a three-character string identifying the version of FrameMaker. For example, the character string `<MakerVector6.0>` identifies an imported graphic created in FrameMaker.

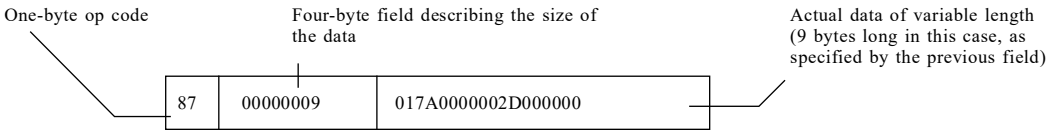
If the imported graphic is stored in a separate file, the file must include the header string `<MakerVectorXXX>`.

Specification of FrameVector data

A description of a graphic in FrameVector format consists of records. Each record contains the following fields:

- A unique one-byte op code
- A four-byte integer specifying the size of the data
- The actual data

The following figure illustrates the breakdown of a typical record:



Types and listing of op codes

Each record begins with an op code. The op code can be one of the following three types:

- Definition
- The definition op codes specify the version of the FrameVector graphic and any global information used in the graphic, such as colors. Any definitions used by the style and object op codes must be specified before these op codes.
- Style

- The style op codes define the styles applied to all operations until the styles are changed. For example, all graphics objects use the same line width, fill pattern, and color until the style op codes change. All styles need to be defined before specifying the first object op code.
- Object
- The object op codes define graphics objects.

The following tables list the op codes, with a brief description of each op code and the number of the page where each op code is described. The definitions of many of these op codes are similar to corresponding MIF statements.

Definition op codes

Op code	Description of op code	Location
0x01	Version number	page 287
0x02	Bounding rectangle	page 287
0x03	CMYK color definition	page 287
0x04	RGB color definition	page 288
0xFF	End of the vector graphics	page 288

Note that the colors defined in a FrameVector graphic can be used only within the FrameVector graphic. These colors cannot be used for other purposes in the document.

If the definition of a color in the FrameVector graphic does not match the definition in the color catalog of the document, FrameMaker uses the definition in the color catalog when displaying the graphic.

Style op codes

Op code	Description of op code	Location
0x06	Dashed line style	page 288
0x07	Arrow style	page 289
0x20	Rotation angle	page 289
0x21	Pen pattern	page 290
0x22	Fill pattern	page 290
0x23	Line width	page 290
0x24	Color	page 290
0x25	Overprint	page 290
0x26	Dashed/solid line	page 291
0x27	Head cap style	page 291
0x28	Tail cap style	page 291
0x29	Smoothed	page 291
0x2A	Font name	page 292
0x2B	Font size	page 292
0x2C	Font style	page 292

Op code	Description of op code	Location
0x2D	Font color	page 293
0x2E	Font weight	page 293
0x2F	Font angle	page 293
0x30	Font variation	page 293
0x31	Font horizontal kerning	page 294
0x32	Font vertical kerning	page 294
0x33	Font word spread value	page 294

Object op codes

Op code	Description of op code	Location
0x80	Ellipse	page 295
0x81	Polygon	page 295
0x82	Polyline	page 295
0x83	Rectangle	page 296
0x84	Rounded rectangle	page 296
0x85	Arc	page 296
0x86	FrameImage graphic imported within this graphic	page 297
0x87	Beginning of text line	page 297
0x88	Text in text line	page 298
0x89	End of text line	page 298
0x8A	Beginning of clipping rectangle	page 298
0x8B	End of clipping rectangle	page 299
0x8C	FrameVector graphic imported within this graphic	page 299

Data types used in specifications

The following table lists the data types used for the specifications in this appendix.

Type	Definition
byte	unsigned 8-bit integer
short	unsigned 16-bit integer
long	signed 32-bit integer
unsigned long	unsigned 32-bit integer
metric	signed 32-bit, fixed point; the first 16 bits represent the digits preceding the decimal, the last 16 bits represent the digits following the decimal
string	string of bytes in UTF-8 encoding
point	2 metrics interpreted as the position of the point in x and y coordinates

Type	Definition
rectangle	4 metrics interpreted as the position of the rectangle in x and y coordinates and the size of the rectangle in width and height

All integer values are stored in big endian order.

The x and y coordinates are relative to the rectangle bounding the vector graphics. The origin of the coordinate system is the upper left corner of this rectangle.

For the specifications of angles, positive values are measured clockwise from 0° (the x-axis), and negative values are measured counterclockwise.

Specifications of definition op codes

This section describes each definition op code. Op codes are listed by number and description. The op code number is shown in parentheses.

Version number (0x01)

Specification by data type:	Byte
Description of data:	Bits 7-4: major version number
	Bits 3-0: minor version number
Size of data in bytes:	1
Example:	01 00000001 50 representing version 5.0
Note:	This must be the first op code for a FrameVector graphic.

Bounding rectangle (0x02)

Specification by data type:	Metric, metric, metric, metric
Description of data:	Position of graphic (metric, metric)
	Width of graphic (metric)
	Height of graphic (metric)
Size of data in bytes:	16
Example:	02 00000010 00000000 00000000 020A0000 00BD0000 for a graphic with the following specifications: x position = 0 points (0000) y position = 0 points (0000) width = 522 points (020A) height = 189 points (00BD)
Note:	This must be the second op code for a FrameVector graphic.

CMYK color definition (0x03)

Specification by data type:	String, metric, metric, metric, metric

Description of data:	Name of color tag (string)
	Percentages of cyan, magenta, yellow, and black (metric, metric, metric, metric)
Size of data in bytes:	Variable
Example:	03 0000001B 00 0B 53 61 67 65 20 47 72 65 65 6E 00 00500000 00230000 00320000 00000000 for a color named Sage Green with the following specifications: cyan = 80% (0050) magenta = 35% (0023) yellow = 50% (0032) black = 0% (0000)
Note:	See "Definition op codes" on page 285 for more information on color definitions.

RGB color definition (0x04)

Specification by data type:	String, metric, metric, metric
Description of data:	Name of color tag (string)
	Percentages of red, green, and blue (metric, metric, metric)
Size of data in bytes:	Variable
Example:	03 0000001B 00 0B 53 61 67 65 20 47 72 65 65 6E 00 00280000 00410000 00330000 for a color named Sage Green with the following specifications: red = 40% (0028) green = 65% (0041) blue = 51% (0033)
Note:	See "Definition op codes" on page 285 for more information on color definitions.

End of the vector graphic (0xFF)

Specification by data type:	N/A
Description of data:	None
Size of data in bytes:	0
Example:	FF 00000000
Note:	This must be the last op code for a FrameVector graphic.

Specifications of style op codes

This section describes each style op code. Op codes are listed by number and description. The op code number is shown in parentheses.

Note that these styles remain in place until another style op code resets the style.

Dashed line style (0x06)

Specification by data type:	Short, metric, ... , metric

Description of data:	Number of dash segments (short)
	Length of dash segments in points (metric, ..., metric)
Size of data in bytes:	Variable
Default value:	None (solid)
Example:	06 0000000A 0002 00080000 00060000 for a dashed line with the following specifications: number of dash segments = 2 dash segment #1 (line segment) = 8.0 points long dash segment #2 (gap in dashed line) = 6.0 points long

Arrow style (0x07)

Specification by data type:	Byte, byte, byte, byte, metric, metric
Description of data:	Tip angle in degrees (byte — between 5 and 85 degrees)
	Base angle in degrees (byte — between 10 and 175 degrees)
	Arrow type (byte — 0:stick, 1:hollow, 2:filled)
	Scale the arrow? (byte — 0:no, 1:yes)
	Length in points (metric)
	Scale factor (metric)
Size of data in bytes:	12
Default value:	default arrow style
Example:	07 0000000C 10 5A 02 00 000C0000 00004000 for an arrow style with the following specifications: tip angle = 16° (10) base angle = 90° (5A) arrow type = filled (02) arrow scaled? = no (00) length = 12 points (000C0000) scale factor = 0.25 (00004000)

Rotation angle (0x20)

Specification by data type:	Metric
Description of data:	Angle in degrees
Size of data in bytes:	4
Default value:	0
Example:	20 00000004 00500000 for the rotation angle of 80°

Pen pattern (0x21)

Specification by data type:	Byte
Description of data:	Index to pen patterns (see "Values for Pen and Fill statements" on page 113)
Size of data in bytes:	1
Default value:	0 (solid)
Example:	21 00000001 00 for a solid pen pattern

Fill pattern (0x22)

Specification by data type:	Byte
Description of data:	Index to pen patterns (see "Values for Pen and Fill statements" on page 113)
Size of data in bytes:	1
Default value:	7 (white)
Example:	22 00000001 07 for a white fill pattern

Line width (0x23)

Specification by data type:	Metric
Description of data:	Width of line in points
Size of data in bytes:	4
Default value:	1 point
Example:	23 00000004 00008000 for the line width of 0.5 point

Color (0x24)

Specification by data type:	String
Description of data:	Name of color tag
Size of data in bytes:	Variable
Default value:	Black
Example:	24 00000006 00 06 42 6C 61 63 6B 00 for the color Black

Overprint (0x25)

Specification by data type:	Byte
Description of data:	Is the object overprinted? (0: no, 1:yes)
Size of data in bytes:	1
Default value:	0 (no)

Example:	25 00000001 00 if not overprinted 25 00000001 01 if overprinted

Dashed/solid line (0x26)

Specification by data type:	Byte
Description of data:	Is the line dashed? (0: no, 1:yes)
Size of data in bytes:	1
Default value:	0 (no)
Examples:	26 00000001 00 for a solid line 26 00000001 01 for a dashed line
Note:	The style of the dashed line is specified by op code 0x06.

Head cap style (0x27)

Specification by data type:	Byte
Description of data:	Style of head cap or line end (0:arrow, 1:butt, 2:round, 3:square)
Size of data in bytes:	1
Default value:	3 (square)
Example:	27 00000001 00 for arrow style

Tail cap style (0x28)

Specification by data type:	Byte
Description of data:	Style of tail cap or line end (0:arrow, 1:butt, 2:round, 3:square)
Size of data in bytes:	1
Default value:	3 (square)
Example:	28 00000001 00 for arrow style

Smoothed (0x29)

Specification by data type:	Byte
Description of data:	Is the object smoothed? (0: no, 1:yes)
Size of data in bytes:	1
Default value:	0 (no)

Example:	29 00000001 00 for an unsmoothed object 29 00000001 01 for a smoothed object

Font name (0x2A)

Specification by data type:	Byte, string, string, string (some strings not used, depending on flag)
Description of data:	Flag indicating which names are used to identify the font (byte — 0:family name, 1:family and PostScript name, 2:family and platform name, 3:all three names)
	Family name (string)
	PostScript name (string)
	Platform name (string)
Size of data in bytes:	Variable
Default value:	default font name
Example:	2A 0000000A 00 00 08 43 6F 75 72 69 65 72 00 for a font specified by the family name Courier

Font size (0x2B)

Specification by data type:	Metric
Description of data:	Point size of font
Size of data in bytes:	4
Default value:	default font size
Example:	2B 00000004 000C0000 for a 12 point font

Font style (0x2C)

Specification by data type:	Unsigned long
Description of data:	Described by 14 bits, where bit 0 is the least significant bit:
	Bit 0: bold (equivalent to setting the font weight to bold)
	Bit 1: italic (equivalent to setting the font angle to italic)
	Bits 2-4: underline style — 0:no underline, 1:single, 2:double, 3:numeric (bit 4 is not currently used)
	Bit 5: overline
	Bit 6: strikethrough
	Bit 7: superscript
	Bit 8: subscript
	Bit 9: outline

	Bit 10: shadow
	Bit 11: pair kern
	Bits 12-13: case — 0:as is, 1:small caps, 2:lower case, 3:upper case
Size of data in bytes:	4
Default value:	default font style
Example:	2C 00000004 00000043 for a font with bold, italic, and strikethrough styles

Font color (0x2D)

Specification by data type:	String
Description of data:	Name of color tag
Size of data in bytes:	Variable
Default value:	Black
Example:	03 0000001B 00 0B 53 61 67 65 20 47 72 65 65 6E 00 for a font in the color Sage Green

Font weight (0x2E)

Specification by data type:	String
Description of data:	Name of font weight type (uses the same values as the MIF `FWeight` statement)
Size of data in bytes:	Variable
Default value:	default font weight
Example:	2E 00000008 00 08 52 65 67 75 6C 61 72 00 for the font weight Regular

Font angle (0x2F)

Specification by data type:	String
Description of data:	Name of font angle type (uses the same values as the MIF `FAngle` statement)
Size of data in bytes:	Variable
Default value:	default font angle
Example:	2F 00000008 00 08 52 65 67 75 6C 61 72 00 for the font angle Regular

Font variation (0x30)

Specification by data type:	String
Description of data:	Name of font variation type (uses the same values as the MIF `FVar` statement)
Size of data in bytes:	Variable
Default value:	default font variation

Example:	30 00000008 00 08 52 65 67 75 6C 61 72 00 for the font variation Regular

Font horizontal kerning (0x31)

Specification by data type:	Metric
Description of data:	Horizontal kerning in percentage on an em (a positive value moves characters to the right, a negative value moves characters to the left)
Size of data in bytes:	4
Default value:	default horizontal kerning
Example:	31 00000004 00008000 for a font kerning of 50% of an em to the right (0.50) 31 00000004 FFFF8000 for a font kerning of 50% of an em to the left (-0.50)

Font vertical kerning (0x32)

Specification by data type:	Metric
Description of data:	Vertical kerning in percentage of an em (a positive value moves characters downward, a negative value moves characters upward)
Size of data in bytes:	4
Default value:	default vertical kerning
Example:	32 00000004 00008000 for a font kerning of 50% of an em downward (0.50) 32 00000004 FFFF8000 for a font kerning of 50% of an em upward (-0.50)

Font word spread value (0x33)

Specification by data type:	Metric
Description of data:	Percentage of spread
Size of data in bytes:	4
Default value:	default word spread
Example:	33 00000004 00008000 for a word spread of 50% (0.50) 33 00000004 FFFF8000 for a word spread of -50% (-0.50)

Specifications of object op codes

This section describes each object op code. Op codes are listed by number and description. The op code number is shown in parentheses.

Ellipse (0x80)

Specification by data type:	Rectangle
Description of data:	Position and size of ellipse in points
Size of data in bytes:	16
Example:	80 00000010 01320000 00240000 007E0000 007E0000 for an ellipse with the following specifications: x position = 306 points (0132) y position = 36 points (0024) width = 126 points (007E) height = 126 points (007E)

Polygon (0x81)

Specification by data type:	Long, point, ..., point
Description of data:	Number of points (long)
	Position of each point in points (point, ..., point)
Size of data in bytes:	Variable
Example:	81 00000010 00000003 01320000 002E0000 01100000 007E0000 01680000 007D0000 for a polygon with the following specifications: number of points = 3 x position of point #1 = 306 points (0132) y position of point #1 = 46 points (002E) x position of point #2 = 272 points (0110) y position of point #2 = 126 points (007E) x position of point #3 = 360 points (0168) y position of point #3 = 125 points (007D)
Note:	When smoothed style is on, this object is a closed Bezier curve.

Polyline (0x82)

Specification by data type:	Long, point, ..., point
Description of data:	Number of points (long)
	Position of each point in points (point, ..., point)
Size of data in bytes:	Variable

Example:	82 0000000C 00000002 00120000 00360000 00FC0000 003F0000 for a polyline with the following specifications: number of points = 2 (00000002) point #1, x position = 18 points (0012) point #1, y position = 54 points (0036) point #2, x position = 252 points (00FC) point #2, y position = 63 points (003F)
Note:	When smoothed style is on, this object becomes a Bezier curve.

Rectangle (0x83)

Specification by data type:	Rectangle
Description of data:	Position and size of rectangle in points
Size of data in bytes:	166
Example:	83 00000010 00670000 004F0000 00130000 003C0000 for a rectangle with the following specifications: x position = 103 points (0067) y position = 79 points (004F) width = 19 points (0013) height = 60 points (003C)

Rounded rectangle (0x84)

Specification by data type:	Metric, rectangle
Description of data:	Radius of corners in points (metric)
	Position and size of rectangle in points (rectangle)
Size of data in bytes:	20
Example:	84 00000014 00120000 007E0000 007E0000 00630000 00240000 for a rounded rectangle with the following specifications: radius of corners = 18 points (0012) x position = 126 points (007E) y position = 126 points (007E) width = 99 points (0063) height = 36 points (0024)

Arc (0x85)

Specification by data type:	Rectangle, metric, metric
Description of data:	Position and size of arc in points (rectangle)
	Start angle in degrees (metric)

	Length of arc in degrees, where positive values correspond to clockwise arcs and negative values correspond to counterclockwise arcs (metric)
Size of data in bytes:	24
Example:	85 00000018 00490000 00270000 007C0000 008C0000 00000000 005A0000 for an arc with the following specifications: x position = 73 points (0049) y position = 39 points (0027) width = 124 points (007C) height = 140 points (008C) start angle = 0° arc angle length = 90°

FrameImage graphic imported within this graphic (0x86)

Specification by data type:	Rectangle, byte, bitmap
Description of data:	Position and size of the bounding rectangle in points (rectangle)
	Is the object flipped left/right? (byte — 0:no, 1:yes)
	FrameImage data (bitmap)
Size of data in bytes:	Variable
Example:	86 00000035 00F20000 00740000 00080000 00080000 00 59A66A95 00000008 00000008 00000001 00000000 00000002 00000000 00000000 80 0E FF 20 for an imported bitmap graphic of a black square with the following specifications: x position = 242 points y position = 116 points width = 8 points height = 8 points flipped left/right = no
Note:	The bitmap is scaled to the size of the bounding rectangle.

Beginning of text line (0x87)

Specification by data type:	Point, byte

Description of data:	Baseline origin of the text line in points (point)
	Text line alignment (byte — 0:left, 1:center, 2:right)
Size of data in bytes:	9
Example:	87 00000009 017A0000 002D0000 00 for a text line with the following specifications: x position = 378 points (017A) y position = 45 points (002D) alignment = left
Note:	The specification of the start of a text line begins with op code 87 and can contain combinations of fonts and text. A text line must end with op code 89.

Text in text line (0x88)

Specification by data type:	String
Description of data:	Actual text written in text line
Size of data in bytes:	Variable
Example:	88 00000005 0005 74 65 78 74 00 for the text line “text”

End of text line (0x89)

Specification by data type:	N/A
Description of data:	None
Size of data in bytes:	0
Example:	89 00000000

Beginning of clipping rectangle (0x8A)

Specification by data type:	Rectangle
Description of data:	Position and size of clipping rectangle in points
Size of data in bytes:	16
Example:	8A 00000010 00670000 004F0000 00130000 003C0000 for a clipping rectangle with the following specifications: x position = 103 points (0067) y position = 79 points (004F) width = 19 points (0013) height = 60 points (003C)

Note:	Clipping rectangles are unique to the FrameVector format. All objects within a clipping rectangle are drawn to the boundaries of the rectangle. If an object extends beyond this region, the portion that passes the rectangle boundary is not drawn. The specification of the start of a clipping rectangle begins with op code 8A and ends with op code 8B. All objects within the clipping rectangle must be specified between these two op codes.

End of clipping rectangle (0x8B)

Specification by data type:	N/A
Description of data:	None
Size of data in bytes:	0
Example:	8B 00000000

FrameVector graphic imported within this graphic (0x8C)

Specification by data type:	Rectangle, byte, vector data
Description of data:	Position and size of the bounding rectangle in points (rectangle)
	Is the object flipped left/right? (byte — 0:no, 1:yes)
	FrameVector data (vector data)
Size of data in bytes:	Variable
Example:	8C 00000046 00670000 004F0000 00130000 003C0000 00 ...(FrameVector data)... for a FrameVector graphic with the following specifications: x position = 103 points (0067) y position = 79 points (004F) width = 19 points (0013) height = 60 points (003C) flipped left/right = no
Note:	The vector graphic is scaled to the size of the bounding rectangle.

Sample FrameVector facet

The sample FrameVector facet in this section describes the following illustration:

This illustration is composed of the following graphic objects:

- A rectangle with no border and a gray fill

- A polygon defined by three points, a black border, and no fill
- A rectangle with a black border and a white fill
- A text line with the text “FrameVector Graphic” in small caps
- A polyline defined by two points and an arrow style head
- An arc with a black border and no fill

The following sample facet describes this graphic.

```
...
=FrameVector
&%v
&<MakerVector6.0>
&\x
&010000000150
&020000001000000000000000000168000000D80000
&230000000400008000
&21000000010F
&24000000080006426C61636B00
&260000000100
&220000000104
&200000000400000000
&8300000010007A00000052000000C0000000190000
&210000000100
&220000000107
&810000001C00000003000E000000410000002900000071000000 4C000000410000
&8300000010007200000 04A000000C0000000190000
&8700000009007B0000005C000000
&2A0000000C00000A\xHelvetica\x00
&2B0000000400090000
&300000000A0008526567756C617200
&2F0000000A0008526567756C617200
&2E0000000A0008526567756C617200
&330000000400008000
&2C0000000400001000
&88000000160014\xFrameVector Graphic\x00
&8900000000
&070000000C10780201000C00000004000
&270000000100
&820000001400000002007200000055000000330000000550000
&22000000010F
&270000000103
&850000001800040000002B0000002F0000002C0000005A0000005A0000
&FF00000000
&\x
=EndInset
...
```

The following sections explain the syntax used to describe this facet.

Definition op codes for the FrameVector graphic

The example begins with the ASCII string `<MakerVector 6.0>`. The `\x` characters indicate that the data that follows is in hexadecimal format.

The following lines specify the FrameVector version 6.0 and the size (5" x 3", or 360 points by 216 points) and position (0,0) of the FrameVector graphic:

```
&010000000150
&020000001000000000000000000168000000D80000
```

Since colors are not used in this example, the color op codes are not specified.

Specification of the rectangle shadow

The drop shadow of the rectangle is drawn first, since it appears behind the other graphic objects. The rectangle has the following specifications:

- The line width is 0.5 point.

```
&230000000400008000
```

- The pen pattern is none (0F).

```
&21000000010F
```

- The color is black.

```
&24000000080006426C61636B00
```

- The line is solid (not dashed).

```
&260000000100
```

- The fill pattern is grey (04).

```
&220000000104
```

- The rotation angle is 0°.

```
&200000000400000000
```

- The position of the rectangle is (122 points, 82 points).

```
&8300000010007A000000520000
```

- The size of the rectangle is 192 points by 25 points.

```
00C0000000190000
```

Specification of the polygon

The polygon in this example has the following specifications:

- The pen pattern is solid (00).

```
&210000000100
```

- The fill pattern is white (07).

```
&220000000107
```

- The polygon has three points.

```
&810000001C00000003
```

- The positions of the three points are (15 points, 65 points), (41 points, 113 points), and (76 points, 65 points).

```
000E000000410000002900000071000000 4C000000410000
```

The rest of the styles are inherited from the previous object.

Specification of the rectangle

The white rectangle in this example has the following specifications:

- The position of the rectangle is (114 points, 74pt).

```
&830000001000720000004A0000
```

- The size of the rectangle is 192 points by 25 points.

```
00C0000000190000
```

The rest of the styles are inherited from previous objects.

Specification of the text line

The text line in this example has the following specifications:

- The position of the text line is (123 points, 92 points), and the text line is left-aligned.

```
&8700000009007B0000005C000000
```

- The text line uses the Helvetica font.

```
&2A0000000C00000A\xHelvetica\x00
```

- The text line uses a 9-point font.

```
&2B0000000400090000
```

- The font variation is Regular.

```
&300000000A0008526567756C617200
```

- The font angle is Regular.

```
&2F0000000A0008526567756C617200
```

- The font weight is Regular.

```
&2E0000000A0008526567756C617200
```

- The font word spread value is 50%.

```
&330000000400008000
```

- The font style is Small Caps.

```
&2C0000000400001000
```

- The text in the text line is "FrameVector Graphic."

```
&88000000160014\xFrameVector Graphic\x00
```

The rest of the styles are inherited from previous objects.

The following record specifies the end of the text line:

```
&8900000000
```

Specification of the polyline

The polyline in this example has the following specifications:

- The arrow style has a tip angle of 16° and a base angle of 120°.

```
&070000000C1078
```

- The arrow style is defined so that the arrow is filled and is scaled as it gets wider. The length of the arrow is 12 points. If the line is widened, the arrow head also is widened by a corresponding factor of 0.25.

```
0201000C00000004000
```

- The style of the head cap of the polyline is arrow.

```
&270000000100
```

- The polyline consists of two points.

```
&820000001400000002
```

- The positions of the two points are (114 points, 85 points) and (51 points, 85 points).

```
00720000005500000033000000550000
```

The rest of the styles are inherited from previous objects.

Specification of the arc

The arc in this example has the following specifications:

- The fill pattern of the arc is none (0F).

```
&22000000010F
```

- The style of the head cap of the arc is square.

```
&270000000103
```

- The position of the arc is (4 points, 43 points).

```
&850000001800040000002B0000
```

- The size of the arc is 43 points by 40 points.

```
002F0000002C0000
```

- The start angle of the arc is 90°, and the arc angle length is 90°.

```
005A0000005A0000
```

The rest of the styles are inherited from previous objects.

Specification of the end of the FrameVector graphic

The following record specifies the end of the FrameVector graphic:

```
&FF00000000
```

The `\x` characters specify the end of data in hexadecimal format.

Chapter 15: Legal notices

For legal notices, visit the Legal Notices page.