Miramo®

Automated Publishing

User Guide

VERSION 9.2

Readers of this documentation should note that its contents are intended for guidance only, and do not constitute formal offers or undertakings.

'License Agreement'

This software, called Miramo, is licensed for use by the user subject to the terms of a License Agreement between the user and Datazone Ltd. Use of this software outside the terms of this license agreement is strictly prohibited. Unless agreed otherwise, this License Agreement grants a non-exclusive, non-transferable license to use the software programs and related documentation in this package (collectively referred to as Miramo) on licensed computers only. Any attempted sublicense, assignment, rental, sale or other transfer of the software or the rights or obligations of the License Agreement without prior written consent of Datazone shall be void. In the case of a Miramo Development License, it shall be used to develop applications only and no attempt shall be made to remove the associated watermark included in output documents by any automated method.

The documentation accompanying this software must not be copied or re-distributed to any third-party in either printed, photocopied, scanned or electronic form.

The software and documentation are copyrighted. Unless otherwise agreed in writing, copies of the software may be made only for backup and archival purposes. Unauthorized copying, reverse engineering, decompiling, disassembling, and creating derivative works based on the software are prohibited. This notice is provided for information only, and does not constitute a License Agreement.

Datazone does not warrant that the software will be free from error or will meet your specific requirements. You assume complete responsibility for decisions made or actions taken based on information obtained using the software. Any statements made concerning the utility of the software are not to be construed as unexpressed or implied warranties.

Trademarks

FrameMaker, FrameMaker+SGML, Acrobat, Adobe, PostScript, Adobe Illustrator and TIFF are trademarks of Adobe Systems Inc. Macintosh is a trademark of Apple Computer Inc. Windows NT is a trademark of Microsoft Corporation.

PANTONE is a registered trademark of Pantone Incorporated.

Miramo is a Registered Trademark of Datazone Ltd (see above).

Miramo includes software developed by the Apache Software Foundation (http://www.apache.org/).

Datazone Limited
Palm Gate, Greenane,
Killarney, Co. Kerry, Ireland.
Tel: +353 64 66 289 64
Fax: +353 64 66 289 65
Email: miramo@datazone.com
www.miramo.com

13.09.17 920p21

User & Reference Guides

Contents

User Guide

Reference Guide

mmDraw Drawing Guide

mmServer Guide

mmChart Inline Charting Guide

Character Reference Guide

User Guide

Contents

CHAPTER 1

Getting started

Miramo produces a formatted document (usually a PDF) from an input stream containing Miramo markup codes (often XML). Miramo markup codes are used to describe or reference the layout and content of the output document.

The Miramo markup code examples given in this chapter can be created manually using a simple text editor. In a production environment the stream of Miramo markup codes is normally generated programatically in response to a user request.

This chapter gives a fast-track introduction to using Miramo markup codes, and gives examples of using the Miramo command line interface (CLI) and/or the Miramo Application Programming Interface (API) to process jobs.

The example files discussed in this chapter are included in the %MM_HOME%\api\testfiles\gettingstarted folder.[1]

Miramo input markup categories

There are two major categories of markup: ***inline*** markup and ***format definitions.*** The application of these markup categories is summarized below.

- ***Inline markup***

  Inline markup is used to describe major document objects, e.g. paragraphs, tables, graphic frames, etc. The following is an example of inline markup codes used to represent an empty table (<Tbl ... >), containing two empty rows (<Row ... >), anchored in a paragraph (<P ... >):

```
<P>
 <Tbl>
  <Row/>
  <Row/>
 </Tbl>
```

  Inline markup is used to achieve special formatting effects, e.g. subscripts, emphasis, markers, etc, for characters and strings of text. Inline markup includes a set of marker codes for inserting text to be used to generate indexes and tables of contents, and variable codes for inserting pre-defined variables (such as the current date, <Date ... />) in paragraph text. Inline markup codes may be placed anywhere where printable paragraph text is allowed.

  A summary of the syntax of inline markup codes is contained in Chapter 3, 'Inline markup codes', in the *Miramo Reference Guide*, pages R-37–40, and an alphabetically ordered description of each inline markup code is contained on pages R-45–297.

1. The MM_HOME environment variable stores the location of the Miramo installation folder.

- ***Format definitions***

  Format definitions are a special category of markup that enable the definition of paragraph formats, table formats, page layouts, etc. Format definitions are an adjunct, or alternative, to using a FrameMaker template document. A FrameMaker template document may be any FrameMaker document created by the same version of FrameMaker as used by mmServer, saved in MIF format. FrameMaker document templates are often used because the FrameMaker GUI provides a quick and easy way to specify complex typographical layouts.

  An alphabetically ordered description of each format definition code is included in Chapter 4, 'Format definitions', in the *Miramo Reference Guide*, pages R-299–413.

  The examples in this chapter assume that a FrameMaker template is used to control the formatting characteristics of the output document rather than Miramo format definition codes.

Simplest input

The simplest possible Miramo input file can have just one line, as shown in Example 1.1. The <P> code here indicates 'start of paragraph'.

```
<P>I didn't steal Lud Moseley's calico horse.
```

Example 1.1 Simplest input (1)

Creating a text file called 'test1' containing the above line and running any of the commands below:

```
miramo -Ofm test1.fm test1
miramo -Pname "Xeikon-12" -Ofm test1.fm test1
miramo -Opdf test1.pdf test1
```

will produce either a FrameMaker document or PDF file containing the single line of text input printed in the default 'Body' paragraph format. The text will be in 12 point TimesNewRoman (with 2 point leading) with a one inch margin on all sides, placed on a US letter sized page (8.5" × 11").

If the second of the three commands above is used, the output will be printed on the 'Xeikon-12' printer. If the third of the three commands above is used, the output will be a PDF document.

The initial '<P>' in the input shown in Example 1.1 is an example of an option-free Miramo markup code. This <P> markup code, like all markup codes, may be terminated by a corresponding termination code, i.e. by a </P> code, as required in XML conformant documents, e.g. documents output via XSLT, as shown in Example 1.3.

```
<P>I didn't steal Lud Moseley's calico horse.</P>
```

Example 1.2 Simplest input (2)

The simplest possible Miramo input must begin with a <P ... > code, which is a container for nearly all other inline markup codes. In addition, every input stream must contain at least one <P ... > code.[1]

1. The exception to this rule is any generated file created using the <GenChapter ... > code to create a table of contents or an index in a book.

Miramo input markup and XML compliance

Miramo input markup may be, but does not have to be, well-formed XML (for example, matching start and end elements, quotes around attribute values) and may, but does not have to, conform to the rules given in the Miramo DTD describing the <MiramoXML ... > root element.

Example 1.3 illustrates an XML compliant version of the Examples 1.1 and 1.2.

The output produced by Examples 1.1, 1.2 and 1.3 is identical.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE MiramoXML SYSTEM "miramo.dtd">
<MiramoXML>
<P>
I didn't steal Lud Moseley's calico horse
</P>
</MiramoXML>
```

Example 1.3 XML compliant input

Using markup code options

More-typical input may look like that shown in Example 1.4.

```
<MiramoXML>
<P fmt="Title" >
Horse Thief
</P>
<P fmt="Author" >
Erskine Caldwell
</P>
<P fmt="BodyText" >
I didn't steal Lud Moseley's calico horse.
People all over have been trying to make
me out a thief. . .
</P>
<P >
. . .
</MiramoXML>
```

Example 1.4 Basic inline markup codes (1)

The <P fmt="Title"> on the first input line in Example 1.4 is an instance of an *inline* markup code with an option and option value. The <P ... > code indicates the beginning of a paragraph. A paragraph may be empty or may contain text and other markup. The 'fmt' option within the <P ... > code indicates that the properties of any subsequent text until the next <P ... > code, or a </P> code, is encountered, will be those of the 'Title' paragraph format, as specified in a template file, or by using the <ParaDef ... >[1] format definition code.

In the most common cases a Miramo document will contain from dozens to thousands of <P ... > codes, many of which will include 'option=value' pairs.[2] A <P ... > code with no options, i.e. <P>, inherits the option values of the preceding <P ... > code.

Option values may be surrounded by single or double quotation marks. The following three forms are identical:

```
<P fmt=Title >
<P fmt='Title' >
<P fmt="Title" >
```

1. The <ParaDef ... > format definition code is described in the *Miramo Reference Guide*, pages R-359–367.

2. A single *document* may contain several million paragraphs and nearly two million table cells. However significant amounts of RAM are needed to process documents of this size. Large documents can often be split into smaller components by using <Chapter ... > codes (see the *Miramo Reference Guide* Chapter 3, pages R-89–104 and pages U-65–75). Doing so will generally result in better performance. If the input stream contains <Doc ... > codes (see the *Miramo Reference Guide* Chapter 3, pages R-113–126), a new document is created for each <Doc ... > code.

There is no realistic limit on the size of a single paragraph: single paragraph sizes in excess of 500 A4/US Letter sized pages comprising 10 on 12 point type may be produced without problem.

Note that the first form is not XML compliant. Spaces and tabs, but not line breaks, are permitted before and after the = (equals sign). Option=value pairs must be separated by one or more space, tab or line break characters.

Nearly all option names may have either a short form or a long form. These forms may be used interchangeably. A complete listing of markup codes and short and long option names is contained in the *Miramo Reference Guide* (see LIST OF INLINE MARKUP CODES AND OPTIONS, starting on page R-541, and LIST OF FORMAT DEFINITION MARKUP CODES AND OPTIONS, starting on page R-561).

Using templates

Assuming a FrameMaker template document[1] called 'formats.mif' has been created containing 'Title', 'Author' and 'BodyText' paragraph format definitions, and if the input lines shown in Example 1.4 are put in a file called 'test2', the following command line:

```
miramo -Tfile formats.mif -Ofm test2.fm test2
```

creates a formatted FrameMaker output file 'test2.fm'. This may be opened and viewed on-screen either within FrameMaker or by typing:

```
mmopendoc test2.fm
```

on a command line.

The formatted output could be as shown in Figure 1.1 at right.

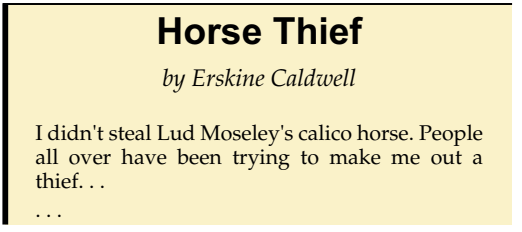


Figure 1.1 Output from Example 1.4

The format of the output shown in Figure 1.1 is a result of using the '-Tfile' option to specify the Miramo template file 'formats.mif' in which the 'Title' paragraph format is defined as centered 14 point Arial bold, and the 'Author' paragraph format is centered 10 point Palatino italic with space below being set to 18 pt. The 'Author' paragraph format also has an 'autonumber' setting which includes the string 'by'. The body text (paragraph tag 'BodyText') is set in 9 point Palatino with 1 point leading.

```
<MiramoXML><P fmt="Title" >Horse Thief</P><P
fmt="Author" >Erskine Caldwell</P><P
fmt="BodyText" >I didn't steal Lud Moseley's
calico horse. People all over have been trying to make
me out a thief ...</P></MiramoXML>
```

Example 1.5 Basic inline markup codes (2)

Rather than specifying a FrameMaker template file using the '-Tfile' command line option, each of these paragraph formats could have been specified using Miramo <ParaDef ... > format definition codes (see Chapter 4, pages R-359–367 in the *Miramo Reference Guide*). Alternatively, the template could be referenced from the input file using the <MiramoXML ... > 'Tfile' option (see page R-452 in the *Miramo Reference Guide*) within the input file.

1. A template document, or file, is a FrameMaker document saved in MIF (Maker Interchange Format).

In Example 1.4 all the markup codes are shown on lines by themselves. This is not necessary. The input shown in Example 1.5 produces identical output to that produced by Example 1.4.

A full description of all command line options is contained in the *Miramo Reference Guide*, Chapter 2, pages R-3–35.

Miramo job processing options and the API

The examples so far in this chapter use the Miramo Command Line Interface (CLI) to process Miramo jobs, for example to produce a PDF file 'test2.pdf' using the 'test2' input file above, referencing format definitions from the FrameMaker template file 'formats.mif' the command line would be:

```
(1) miramo -Tfile formats.mif -Opdf test2.pdf test2
```

Miramo also incorporates the IJobProcessor interface class (C++) or JobProcessor class (.NET, Java) within the Miramo API which provides equivalent functionality to the command line interface (CLI) described in this chapter.

Here are the equivalent API function calls for (1) above, shown using the C++ API:

```
C++     pJobProcessor->setTemplateFile(L"formats.mif");
        pJobProcessor->createPdf(L"test2", L"test2.pdf");
```

The .NET and Java APIs provide identical functionality:

```
C#      jobProcessor.setTemplateFile(@"formats.mif");
        jobProcessor.createPdf(@"test2", @"test2.pdf");

Java    jobProcessor.setTemplateFile("formats.mif");
        jobProcessor.createPdf("test2", "test2.pdf");
```

It is possible to request printing services and, concurrently, save in a range of different formats. The following (single) command line:

```
miramo -Tfile formats.mif -Ofm test2.fm -Pname "Xeikon-12" -Preg -Pcopies 2 -Pscale 50
       -Opdf test2.pdf test2
```

concurrently saves the output in a FrameMaker document file and a PDF, as well as printing two reduced-size copies with registration marks on the "Xeikon-12" printer. Here is the equivalent set of API function calls using the C++ API:

```
C++     pJobProcessor->setTemplateFile(L"formats.mif");
        pJobProcessor->setOption(L"Ofm", L"test2.fm");
        pJobProcessor->setOption(L"Pname", L"Xeikon-12");
        pJobProcessor->setOption(L"Preg");
        pJobProcessor->setOption(L"Pcopies", L"2");
        pJobProcessor->setOption(L"Pscale", L"50");

        pJobProcessor->setOption(L"Opdf", L"test2.pdf");
        pJobProcessor->run(L"test2");
```

Note that the last two lines above could be replaced by a single call to createPdf() as follows:

```
pJobProcessor->createPdf(L"test2", L"test2.pdf");
```

To pass the entire job processing request at once, the executeMiramoRun() method may be used, for example:

```
pJobProcessor->executeMiramoRun(L"-Tfile formats.mif -Ofm test2.fm -Pname "Xeikon-12"
        -Preg -Pcopies 2 -Pscale 50 -Opdf test2.pdf test2");
```

Basic markup codes

The example of markup coding at the beginning of the previous section contained a single input line. Example 1.6 below introduces referencing a template file from within the input data stream, tagged and untagged paragraph codes with override options, as well as font formatting and special *character* codes.

```
1  <!-- This is a comment -->
2  <MiramoXML Tfile="mmuserguide.mif" >
3  <P fmt="output">
4  Hello world
5  </P>
6  <P fmt="output" li="1.0cm" fi="1.0cm">
7  <Font fw="Bold">Viva<Font p="12.1"
8  fa="italic">Valparaiso!</Font><br/>
9  &iexcl;Viva Valparaiso!
10 </P>
11 <P>
12 It's a great city, by the
13 </P>
14 <P p="14" >
15 sea
16 </P>
17 <P fi="1.01415in" >
18 shore.
19 </P>
20 <P fmt="output">Viva Valparaiso</P>
21 <P fmt="body" ><Font fw="Bold">Ol&eacute;</P>
```

Example 1.6 Inline and character markup codes

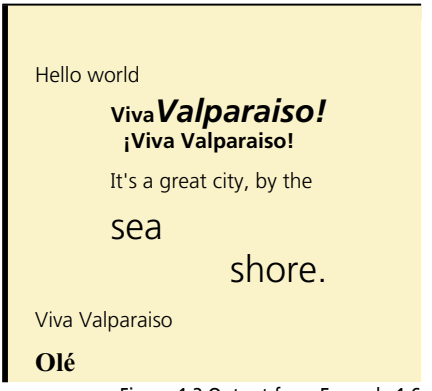


Figure 1.2 Output from Example 1.6

The output from Example 1.6 above could be formatted as shown in Figure 1.2 above.[1]

The <MiramoXML ... > code 'Tfile' option on line 2 in Example 1.6 requests that Miramo include text and other format definitions from a FrameMaker document file called 'mmuserguide.mif', located in the current directory or folder. If no 'Tformats' or 'TXformats' options are included within the <MiramoXML ... > code then *all* object definitions are copied from the template document, as above.

Alternatively the second line in Example 1.6 may be omitted if the template file is specified on the Miramo command line, using the '-Tfile' option:

```
miramo -Tfile mmuserguide.mif -Ofm viva.fm viva
miramo -Tfile mmuserguide.mif -Tformats p -Ofm viva.fm viva
```

The '-Tformats' command line option enables selection of specific object definitions to copy from the template file (see pages R-20–20). In the second example, the '-Tformats p' option is used to copy only paragraph format definitions from the template file. In general using the '-Tfile' command line option is more flexible than including a reference to a

1. The precise appearance will depend on the fonts used for the 'output' and 'body' paragraph formats. Similar dependencies are assumed silently throughout the remainder of this documentation.

template file in the input data. A template file referenced using the <MiramoXML ... > '-Tfile' code will override the definitions of objects having the same names in a template specified using the '-Tfile' command line option.

The fourth line in Example 1.6 contains a tagged paragraph code, '<P fmt="output" >'. In the FrameMaker template file, 'mmuserguide.mif', 'output' is a paragraph format specifying two points of leading and eight point Frutiger-Light. Formatted output produced using Miramo will use this paragraph specification to format the following text. Changing the specification of 'output' in the 'mmuserguide.mif' template file will change the format of the document produced by Miramo.

Options may be added to most inline markup codes. The markup on line 6 in Example 1.6,

```
6   <P fmt="output" li="1.0cm" fi="1.0cm">
```

is a request to left indent the text by 1.0 cm for this instance of an 'output' paragraph.[1]

A following <P ... > code without a tag name and without any overrides inherits all the properties of the preceding paragraph (line 11). Untagged <P ... > codes can also have overrides, as shown in line 14 in Example 1.6.

The next time a '<P fmt="output">' is used without overrides there will be no indentation, as shown (line 20 in Example 1.6 on page U-6).

Note that paragraph text following an untagged <P ... > code inherits the formatting options of the immediately preceding paragraph (see input lines 14 and 17 and their corresponding output).

Nearly sixty <P ... > code options are listed in the *Miramo Reference Guide* (see pages R-220–230). Any document design may be produced using just one paragraph format, creating variations by adding options as required. However this should be avoided as it will result a poorly structured Miramo file which will be difficult to maintain. Most <P ... > code options are intended to be a convenience feature.[2]

If a user changes the definition of the 'output' paragraph format in the FrameMaker template file, the changes will be reflected in the output, provided that copying of paragraph formats is enabled using the '-Tfile' or '-Cfile' command line flags (see pages R-19–19 in the *Miramo Reference Guide*), or the corresponding <MiramoXML ... >, <Doc ... >, <Chapter ... > and <GenChapter ... > code options.

1. If 'cm' (centimeter) is omitted, from the 'li' (left indent) and 'fi' (first indent) options, the value will default to points or to the default document units: alternatively different unit values may be specified in place of cm, e.g. 'in'.

2. Exceptions to this include the 'nl' and 'PSF' options (see the *Miramo Reference Guide*, pages R-227–227). These options cannot be set as paragraph format defaults.

Text attribute codes

```
6  <P fmt="output" li="1.0cm"
       fi="1.0cm">
7  <Font fw="Bold">Viva<Font
       p="12.1"
8  fa="italic">Valparaiso!</Font><br/>
9  &iexcl;Viva Valparaiso!
```

Example 1.7 Lines 6 through 9 from Example 1.6.

Lines 6 through 9 in Example 1.6 (repeated in Example 1.7, from page U-6) contain text attribute and special character codes. The '<Font fw="Bold">' and '<Font fa="italic">' codes tell Miramo that the following text should be printed in bold and italic type respectively. These options are switched off by '</Font>'. The character code '¡' includes an inverted exclamation mark, i.e. '¡', in the formatted output.

Attribute	Start	End
Superscript	<Font u="Y">	</Font>
Subscript	<Font d="Y">	</Font>
Underline	<Font L="S">	</Font>
Double underline	<Font L="D">	</Font>
Small caps	<Font fc="S">	</Font>

Figure 1.3 Examples of changing text attributes

In-text markup codes are cumulative. A '<Font fw="Bold">' followed by a '<Font fa="Italic">' without a preceding '</Font>' results in bold-italic text. Input text containing '<Font fw="Bold">Bonjour</Font>' will print as '**Bonjour**'. '<Font p="12.1">' is an instruction to change the current font size to 12.1 points. Table 1.3 lists additional examples of text attribute codes. The full set of font markup codes is presented in the *Miramo Reference Guide* on pages R-137–142.

Input processing: comments, tabs and blank lines

Comments in the input are delimited with the '<!--' and '-->' markup codes. Comments may appear anywhere in the input, and may be nested. This is illustrated in Example 1.8.

```
<Cell br="Thick" tr="Thick" >
<!-- Product name -->
Murtle bookcases
<Cell br="Thick" tr="Thick" >
<!-- Product description -->
White or blue/birch colored
edging. 4 adj. shelves ...
<!-- This is a comment
<!-- with another comment
nested within it --> -->
```

Example 1.8 Including comments

Blank input lines are ignored. This is illustrated in Example 1.9.

```
<P fmt="output">
People all over have been trying to make me out a thief,

but anybody who knows me at all

will tell you that I've never been in trouble like this in all my life.
```

Example 1.9 Empty input lines

The output from Example 1.9 is shown in Figure 1.4. By default binary tab characters in the input are treated the same as the <tab/> inline markup code. Such binary tabs can be ignored by using the '-stripInput' command line option, described in the *Miramo Reference Guide* on page R-8.

People all over have been trying to make me out a thief, but anybody who knows me at all will tell you that I've never been in trouble like this in all my life.

Figure 1.4 Output from Example 1.9

Spaces at the ends of input lines

As a general rule a single space is silently added to the end of all text lines which do not terminate with a space.

```
1  <P fmt="output">
2  <!-- Convert <FontRef fmt=F_NoLang >&#x2018;</FontRef>didn't&#x2019; into small caps -->
3  I <Font fc="S">didn't</Font> steal Lud Moseley's calico horse.
4
5  <P fmt="output">
6  People all over have been trying to make me out a thief,
7  <FNote>
8  From <Font fa="Italic">Horse Thief</Font>, a short story by Erskine Caldwell.
9  </FNote>
10 but
11 anybody who knows me at all will tell you ...
```

Example 1.10 Adding spaces, and footnotes

Example 1.10 will format as shown in Figure 1.5 Note that no space is added to a line ending immediately preceding an occurrence of a <FNote ... > code (see line 7 in Example 1.5).

I DIDN'T steal Lud Moseley's calico horse.

People all over have been trying to make me out a thief,[1] but anybody who knows me at all will tell you ...

1. From *Horse Thief*, a short story by Erskine Caldwell.

Figure 1.5 Output from Example 1.10

Footnotes

Footnotes may be included within body text and within tables using the <FNote ... > code (see pages R-132–136 in the *Miramo Reference Guide*). The special formatting requirements of both body text footnotes and table footnotes, specifically the clear space above and separator rules, if any, can be controlled using a named frame on a template file reference page. Other properties, e.g. referencing style, sequencing and positioning, are set via the document formatting options in a template file. Alternatively all footnote properties can be defined programatically using the <PageDef ... >, <Frame ... >, <ParaDef ... > and <DocDef ... /> format definition codes.

Footnotes referenced within body text are placed at the bottom of the current page or, if there is insufficient room remaining on the current page, at the bottom of the next page. Footnotes in tables are always placed at the end of the table.

Including markup codes in printed text

A markup code may be included as printable text in the output document by using the <, > and & special character codes in place of '<', '>' and '&' characters respectively. This is illustrated by reference to Example 1.11, which contains the literal input required output these special characters. The output from Example 1.11 will be as shown in Figure 1.6.

```
<P>
The &lt;P&gt; code is used to begin a paragraph.
The &amp;lt; code in the above is printed as '&lt;'
```

Example 1.11 Including markup codes as printable text

The <P> code is used to begin a paragraph.
The < code in the above is printed as '<'

Figure 1.6 Output from Example 1.11

Simplifying data presentation using pre-processors

In most practical applications using *mmpp*, the built-in Miramo macro pre-processor or *mmxslt*, the built-in XSLT transformation engine, provides at least two major benefits. The first major benefit is that presentation of data for processing by Miramo can be greatly simplified. The second major benefit of using *mmpp* or *mmxslt* is that they provide support for implementing complex formatting logic: *mmpp* and *mmxslt* can check data values in the input stream and use these values to change any aspect of the document layout based on programmatic rules. *mmxslt* enables documents to be loaded into memory and re-scanned. *mmpp* is simpler and faster than *mmxslt*.

As a general rule only one pre-processor should be used. *mmxslt*, which has several powerful, Miramo-specific enhancements, e.g. for image processing, or any other XSLT processor may only be used if the input is in XML format. For non-XML input, always use *mmpp*, which is very efficient and has specialized image processing and file reading and writing functions.

Simplifying data presentation using *mmpp*

A detailed description of *mmpp* is contained in the *Miramo Reference Guide*, Chapter 6 (pages R-463–529).

At the simplest level *mmpp* enables the grouping of Miramo markup codes in almost any way. For example, the input shown in Example 1.12 (repeated from Example 1.4 on page U-3), might more easily be extracted from a database in the form shown in Example 1.13

```
<P fmt="Title">
Horse Thief
</P>
<P fmt="Author">
Erskine Caldwell
</P>
<P fmt="BodyText">
I didn't steal Lud Moseley's calico horse.
People all over have been trying to make
me out a thief. . .
</P>
<P>
. . .
```

Example 1.12 Sample input (from Example 1.4)

```
<titlePage title="Horse Thief"
      author="Erskine Caldwell" />
<Para>I didn't steal Lud Moseley's calico
horse.  People all over have been trying to
make me out a thief. . . </Para>

. . .
```

Example 1.13 Simplified input using *mmpp*

The procedure for simplifying input as shown above is itself simple. All that needs to be done is define two macros, as shown in Example 1.14.

```
<@xmacro> titlePage {
        @xgetvals()
        <P fmt=Title><#title></P>
        <P fmt=Author><#author></P>
}
<@xmacro> - Para {
        @xgetvals()
        }{
        <P fmt=BodyText>$Para</P>
}
```

Example 1.14 Simple macro definitions

include these macro definitions at the beginning of the data input stream, and run

Miramo using the '-M' option.[1]

Processing XML data using *mmxslt*

In the special case of processing files or input datastreams encoded in XML it is usually most convenient to use the *mmxslt* pre-processor.

mmxslt can be used as a in-line filter, as shown in Example 1.15 below.

```
cat xmlfile.xml ¦ mmxslt -Xxsl xslfile.xsl ¦ miramo -Opdf pdffile.pdf
```

Example 1.15 Running *mmxslt*

The XSL stylesheet should have an output encoding set to "ISO-8859-1" or "ASCII".

mmxslt is a modified version of the Apache Xalan XSLT engine. Type:

```
mmxslt -help
```

to see a list of the command line options.

More information about using Miramo to format XML encoded data is contained in Chapter 5, *Miramo DTD and XML codes,* on pages R-415–461 in the *Miramo Reference Guide*.

Several guides to XSLT have been published in book form. One example is *XSLT* by Doug Tidwell, O'Reilly and Associates, 2001, Sebastopol, ISBN 0-596-00053-7.

Filename extensions

The filename extensions '.mif' and '.fm' used in the preceding examples are conventions, and are not required. Additional conventions for filename extensions are: '.mm' for a Miramo input file (not used in many of the examples above), '.mif' for a FrameMaker document template, which is a special case of a MIF file,[2] , '.mmp' for a file containing *mmpp* macro definitions, and '.xsl' for a file containing an XSLT stylesheet.

Separating form and content

Sparing use of options with the <P ... > and <Tbl ... >[3] inline markup codes often pays big dividends with regard to ease of document production and maintenance.

The <P ... > inline markup code supports nearly sixty options. Exact equivalents for nearly all of these options exist in the FrameMaker 'Paragraph Designer', as well as in the Miramo <ParaDef ... > format definition code. There are additional options which control the use of different page layouts, or 'master' pages.

To maintain maximum separation between 'form' and 'content', the use of <P ... > codes in Miramo input files should be restricted to have no options. Only the <P ... > code 'fmt' option should be used, without override options. In this case the appearance of paragraph text may be wholly controlled either by editing the settings in the 'Paragraph Designer' in

1. An alternative, more generalized, approach is to include all *mmpp* macro definitions in one or more special files and use the Miramo '-Mfile' command line option.

2. A FrameMaker template may be any FrameMaker document saved in MIF format.

3. The <Tbl ... > inline markup code is discussed in detail in the next chapter.

the template file, if any, or by editing <ParaDef ... > format definitions, or by a combination of both these methods.

Similarly text markup of the form <Font fw="Bold">*xyz*</Font> to request bold text, or similar text markup to request underlines, overlines, italics, etc., should be avoided, if it is desired to maintain maximum separation between form and content. In these cases the usage <Font fmt="e1">xyz</Font>, where *'e1'* is the tag name of an emphasis type defined in the 'Character Designer' of a template file, or the tag name of a <FontDef ... /> format definition, is preferable.

Character sets and encodings

All characters in the Unicode Basic Multilingual Plane are supported, provided the corresponding glyphs are encoded in the current font. Additional characters may be included using the <MapChar ... > code (see pages R-342–346 in the *Miramo Reference Guide*).

By default the input character encoding is assumed to be binary UTF-8. Internally Miramo processes characters in UTF-8 encoding. All characters may additionally be represented by numeric character references in decimal or hexadecimal format. For example, the LATIN CAPITAL LETTER A, U+0041, and the LEFT SINGLE QUOTATION MARK, U+2018, may each be represented in three different forms: binary UTF-8 (41 in hexadecimal), A, A and binary UTF-8 (e28098 in hexadecimal), ‘, ‘.

```
<P fmt="Body" ff="Arial" p="7pt" nl="Y" >
-----------------
U+003C  (<) 3c      &lt;     &#x3C;     &#60;
U+003E  (>) 3e      &gt;     &#x3E;     &#62;
U+0026  (&) 26      &amp;    &#x26;     &#38;
-----------------
U+2018  (') e28098  &lsquo;  &#x2018;   &#8216;
U+2019  (') e28099  &rsquo;  &#x2019;   &#8217;
-----------------
<abc/>
</P>
```

Example 1.16 Alternative character representations

A special set of characters may represented in a fourth format, i.e. as character entities. The set of built-in character references includes: &, <, >, " and '.

Example 1.16 shows possible representations for an illustrative set of characters and text strings. The output from Example 1.16 is shown in Figure 1.7.

```
-----------------
U+003C (<)  3c      <      <      <
U+003E (>)  3e      >      >      >
U+0026 (&)  26      &      &      &
-----------------
U+2018 (')  e28098  ‘      ‘      ‘
U+2019 (')  e28099  ’      ’      ’
-----------------
<abc/>
```

Figure 1.7 Output from Example 1.16

More information about built-in character entities, along with a complete listing, is contained in the *Miramo Character Reference Guide* (see pages CH-3–16). An alphabetical listing of the built-in character entities is shown in Appendix 1:, Index of built-in character entities, on page CH-421 in the *Miramo Character Reference Guide*.

Additional character entity references may be user-defined using the <MapChar ... > code (see pages R-342–346 in the *Miramo Reference Guide*).

If an input string beginning with & and ending with ; is encountered it is replaced provided there is a corresponding entity definition, either built-in or user-defined. If there is no corresponding entity definition the entire string is output as literal text, with a warning message.

CHAPTER 2

Tables

Many types of information are best presented in tables. Such information is usually text and numerical data subject to logical arrangement in rows and columns. Tabular formats can sometimes be achieved using tabs (and in some special circumstances this is the best way). Normally it is easier and more effective to use special table markup codes.[1]

Superior table formatting is a strong feature of Miramo: highly complex and precise table layouts may be undertaken with confidence. Tables may also be enormously long.

Simple tables may be produced using simple markup. However learning to produce complex tables, or even simple tables with specialized typographic features, requires some effort. The best way to acquire Miramo table formatting skills is through trial and error. The examples shown in this chapter may be used as models as a basis for modification and adaptation.

Text data in tables can be formatted in almost any way that is possible outside tables. Table cells may be filled with patterns or colors, and their contents may be rotated. In addition to text data, tables can include anchored frames, images and imported reference frames. Finally tables may be used in unexpected ways to achieve special formatting effects for non-tabular data, e.g. placing images precisely and emphasizing text blocks with adjacent rules.

This chapter contains many examples of using tables. In some cases significant programming time saving will be made by encoding table markup in *mmpp* macros. An example of how to do this is shown later in this chapter, on pages U-27–29.

Basic table markup rules

All tables must be placed within paragraphs, i.e. at least one <P ... > code, optionally followed by text, must precede the first table in a document.[2]

Every table must begin with a <Tbl ... > code, must contain at least one body <Row ... > code and must be terminated by a </Tbl> code. A <Tbl ... > code may incorporate any of nearly thirty <Tbl ... > code override options (see the detailed description of the <Tbl ... > code in the *Miramo Reference Guide*, Chapter 3, pages R-251–263). The 'fmt' option (table format tag) can be a reference to a table format defined in a template file or a reference to a table format defined using <TblDef ... >, or a reference name for a table format which is undefined.

The table code may immediately be followed by one or more <TblColWidth ... /> codes, which set the number and widths of the table columns. See the description of the

1. The main drawback of using tables is that an instance of a table row cannot be split across text frames or pages.

2. A <Tbl ... > code may also immediately follow a <Cell ... > code (which inserts a 'hidden' paragraph in the table cell).

<TblColWidth ... /> code in the *Miramo Reference Guide* on page R-269. If no <TblColWidth ... /> codes are specified, the number and widths are taken from the given table format, or are set to three columns of widths 38mm, 19mm and 19mm if no column widths are available.

Minimal tables

Four illustrations of minimal, dataless table markup are shown in Example 2.1.

```
<Tbl>                                   <!-- 1st table -->
<Row/>
</Tbl>

<Tbl fmt=None >                         <!-- 2nd table -->
<TblColWidth W=2cm/><TblColWidth W=4in />
<Row/>
</Tbl>

<Tbl fmt="Current Costs">               <!-- 3rd table -->
<TblColWidth count=5 W=19mm />
<Row/>
</Tbl>

<!-- 4th table -->
<Tbl fmt="Current Costs" W=95mm n=5 >
<Row/>
</Tbl>
```

Example 2.1 Minimal, dataless table markup

The first table in Example 2.1 contains no table format or tag name, and the table will therefore have the default Miramo table format. This format is described later in this chapter, in the section headed **Default table format** on pages U-17–18.

In the second table in Example 2.1 the name of the table format is 'None'. If the table format is non-existent[1] then the name string is simply a reference name, and the general appearance of the table may be defined using the <Tbl ... > options listed in the *Miramo Reference Guide* on pages R-251–263. If a <Tbl ... > code has a 'fmt' option with a value that does not correspond with a defined table format, and the <Tbl ... > code does not include the 'n' option to specify the number of columns, and no <TblColWidth ... /> code is used, then the number of columns defaults to 3.

The third and fourth tables in Example 2.1 are equivalent.

Summary of table markup sections

Tables may contain six distinct markup elements, or sections, as listed below.

1. ***Table format section***, comprising a <Tbl ... > or a <TblFrame ... > code, optionally followed by one or more <TblColWidth ... /> codes. Every table must contain a table format section.
2. ***Table title section***, beginning with a <TblTitle ... > code (see the *Miramo Reference Guide*, pages R-279–280) and terminated by a </TblTitle>, or a <TblColFormat ... /> or <Row ... > code.
3. ***Table header section***. Any <Row ... > code with a 'type' option set to H will be placed in the table heading. The table header section is optional.

1. The table format name 'None' itself has no special significance: there is no prohibition against defining a table format and giving it a tag name of 'None'. The tag 'None' is used here simply as an example of a table tag which does *not* correspond with any defined table format (i.e., a table format not defined either in Miramo template file, or using the <TblDef ... > format definition code).

4 ***Table body section***, beginning with a single <Row ... > code and optionally followed by several (or hundreds, or thousands of) additional <Row ... > codes. Every table instance must contain a minimum of one body row.

5 ***Table footer section***. Any <Row ... > code with a 'type' option set to F will be placed in the table footing.

6 ***Table termination code.*** A </Tbl> code or, if the table is initiated by a <TblFrame ... > code, a </TblFrame> code, at the end of the table.

A table <Row ... > code may be preceded by one or more <TblColFormat ... /> codes. A table beginning with a <TblFrame ... > code also supports placing the <mmDraw> and <FrameImage ... /> codes ahead of the first <Row ... > code.

Mandatory table markup

Illustrative input for a table containing just the three mandatory sections is shown in Example 2.2.

```
<P>
<!-- Table format section -->
<Tbl fmt="Format A" >
<TblColWidth W=2.8cm/> <TblColWidth W=4cm/>

<!-- Table body section -->
<Row><Cell>Tolstoy, Leo<Cell>The Kreutzer Sonata
<Row><Cell>Ibsen, Henrik<Cell>The Dolls House

<!-- Table termination -->
</Tbl>
```

Example 2.2 Table containing basic markup sections

The output from running Example 2.2 in conjunction with a template containing the standard 'Format A' table definition is shown in Figure 2.1.

Tolstoy, Leo	The Kreutzer Sonata
Ibsen, Henrik	The Dolls House

Figure 2.1 Output from Example 2.2

In Example 2.2 the <Tbl ... > code 'fmt' option setting, 'Format A', specifies that all the options defined in the Frame Table Designer for the table type 'Format A' should be applied to this table, assuming that 'Format A' is a defined table in a template file and that Miramo is run using the '-Tfile *filename*' command line option or, alternatively, that 'Format A' is defined using a <TblDef ... > format definition code.[1]

The format options of a defined table format may be overridden for any table instance by using options following the <Tbl ... > code. For example, if 'Format A' specifies that the table is left aligned by default, setting the <Tbl ... > code 'A' option to R will ensure that the current table is right aligned.

The number of <TblColWidth ... /> codes dictate that this instance of table 'Format A' shall have two columns, having widths of 2.8 cm and 5 cm respectively.

The table in created in Example 2.2 and shown in Figure 2.1 is drawn with rules around the outside of the table and between rows and columns since that is what is specified by

1. If the table type 'Format A' is defined both in the template file and using the <TblDef ... > code, then the options specified by the <TblDef ... > code override settings in a FrameMaker template document. The <TblDef ... > code is described on pages R-391–397.

'Format A'. These rules can be changed or removed either by creating a new table format, or by changing the specification of 'Format A' or by using temporary overrides for this table instance only (see the *Ruling options* listing for the <Tbl ... > code, on page R-255). Similarly, the sizes of the table cell margins may be changed selectively using the <Tbl ... > 'clm', 'ctm', 'crm' and 'cbm' options (see page R-254, and the section **Table cell margins and table cell borders** on page U-25 later in this chapter). By default the height of the rows containing text only is determined by the top and bottom table cell margins, plus the size and leading of the text font, plus any additional top or bottom cell margin space specified for the paragraph.

Header, footer and title sections

The previous table markup (in Example 2.2) is extended to include all six markup sections in Example 2.3. It is also extended to show how to use the <Row ... > and <Cell ... > codes, along with <P ... > codes, within a table cell to override the default table cell paragraph format.

```
1       <P><Tbl fmt="Format A" >
2       <TblColWidth W=31mm /><TblColWidth W=35mm />
3       <TblTitle>Moral Tales</TblTitle>
4       <!-- Next row is 1st & only heading row  -->
5       <Row type=H ><Cell>Author</Cell><Cell>Title</Cell></Row>
6
7       <Row>                   <!-- First body row -->
8       <Cell>Tolstoy, Leo</Cell> <!-- uses 'CellBody' paragraph format -->
9       <Cell>
10      <P fmt=StoryTitle >The Kreutzer Sonata</Cell></Row>
11      <Row>
12      <Cell>                  <!-- Uses 'CellBody' paragraph format -->
13  Ibsen, Henrik</Cell><Cell><P fmt=PlayTitle >
14  The Dolls House</Cell>
15      </Row>
16      <!-- Next row is 1st & only footing row  -->
17      <Row type=F >           <!-- (uses 'CellHeading' default paragraph format) -->
18      <Cell>Footer cell1</Cell><Cell>Footer cell2</Cell></Row>
19      </Tbl>
```

Example 2.3 Table showing all markup sections

The output from Example 2.3 is shown in Figure 2.2.

Table 1: Moral Tales

Author	Title
Tolstoy, Leo	The Kreutzer Sonata
Ibsen, Henrik	*The Dolls House*
Footer cell1	Footer cell2

Figure 2.2 Output from Example 2.3

The title, 'Moral Tales', is centered because that is the alignment specification of the 'TableTitle' paragraph format, which is the default table title paragraph for the 'Format A' table definition. The title is placed above the table because this placement is specified by the 'Format A' table specification. The table footer text is centered within each cell because this is the alignment specified by the 'CellHeading' paragraph format. This can be overridden on a per cell basis using the <Cell ... > 'A' option.

Table header, body or footer sections may be included in any order. If one or more footer rows (<Row ... > code with 'type' option set to F) is included before one or more table header rows (<Row ... > code with 'type' option set to H) or table body rows, the output will be the same as if these rows types were included in groups in their natural order.

Data contained in table headers and footers is repeated on output from one page to the next until the table end.

Default table format

A default table format, named '~mmT', is used whenever a <Tbl ... > code does not include a 'fmt' option specifying the table format name, or if no definition is available for the specified format name. This default table format contains three columns (38 mm, 19 mm and 19 mm wide), one heading row, two body rows and one footing row, and includes a table title. The ~mmT table format is shown in Figure 2.3.

Table 1:

Figure 2.3 Built-in default table format

Figure 2.3 shows the table in exactly the form it would have if it were to be manually inserted into an on-screen FrameMaker document.

The text strings beginning with '~' in Figure 2.4 indicate the names of the default paragraph formats used in each column in the table header, body and footer sections. These paragraph formats, as well as two ruling formats, are always included in documents produced by Miramo if these documents contain one or more tables. Notice that the paragraphs are left aligned in the first two columns, and right aligned in the third column. The <TblColFormat ... /> code may be used to modify or set default paragraph and cell options for columns within a table (see pages R-264–266 in the *Miramo Reference Guide*).

Table 1: ~mTT

~mP_CH	~mP_CH	~mP_CH
~mP_CB	~mP_CB	~mP_CB
~mP_CB	~mP_CB	~mP_CB
~mP_CF	~mP_CF	~mP_CF

Figure 2.4 Built-in default table format (showing default paragraph formats)

Two default rulings are defined for the table. '~mR_0.2pt', a black single rule 0.2 points thick, is used for all columns and between body rows. '~mR_0.8pt', a black single rule 0.8 points thick, is used above and below all header and footer rows.

The default left, top, right and bottom cell margins for the '~mTT' table format are 5, 4, 4 and 2 points respectively. The precise details of how text fits into the body cells of the default table format are shown in Figure 2.5.

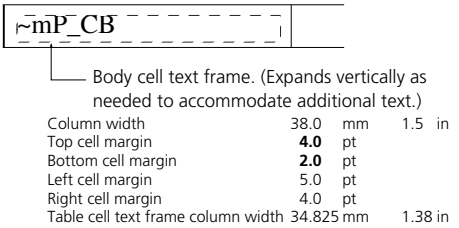


Figure 2.5 Default table body cell layout

Note that the width of the text frame, indicated by the dashed outline, is 9 points (3.175 mm) narrower than the width of the table column. Similarly, the text frame starts 4 points below the top of the cell and finishes two points above the bottom of the cell.

Figure 2.6 shows the equivalent detail for header cells. All is the same as for body cells except that the top and bottom margins are each 2 points bigger.

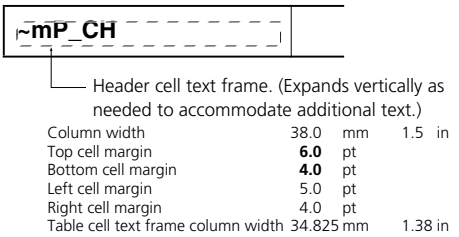


Figure 2.6 Default table header cell layout

Cell text formatting and horizontal alignment

By default the horizontal alignment of text in tables is the same as the alignment settings of the paragraphs used within each table cell. The default cell and paragraph options used for a table column may be set by placing one or more <TblColFormat ... /> codes ahead of the <Row ... > code.

The paragraph and cell settings specified by the <TblColFormat ... /> codes remain in effect through all rows until the next <TblColFormat ... /> code is encountered. However local overrides within a row can be achieved by specifying cell options on the <Row ... > and <Cell ... > codes, and paragraph options on the <Cell ... > or <P ... > codes, all of which have a higher precedence than those specified using a <TblColFormat ... /> code.

The <P ... > paragraph options have a higher precedence than the <Cell ... > paragraph options. Similarly, the <Cell ... > cell options have higher precedence than the <Row ... > cell options. This enables fine control over groups of cells, paragraphs and rows.

The default paragraph and cell options for an entire table may be set by placing a <TblColFormat ... /> code before the first row:

```
<P fmt=P_Tbl>
<Tbl n=4 fmt="Rivers" >
<TblColWidth W=2.0cm />
<TblColWidth W=1.5cm />
<TblColWidth W=1.0cm />

<TblColFormat fmt=P_Rivers A=L />
<!-- First heading row (straddled) -->
<Row type=H fmt="P_Rivers" bold=Y br="R_VThin">
    <Cell cs=4>Long Rivers of the World</Cell>
</Row>

<!-- Second heading row (uses fmt=P_Rivers) -->
<Row type=H >
    <Cell>Location</Cell>
```

```
16        <Cell>Name</Cell>
17        <Cell>km</Cell>
18        <Cell>Miles</Cell>
19    </Row>
20    <!-- Draw red line by filling 1.5 pt body row -->
21    <Row type=H h=1.5pt fill=0 color=Red wp=Y>
22
23    <!-- First body row -->
24    <Row><Cell>Africa<Cell>Nile<Cell>6,695<Cell>4,155</Row>
25    <Row><Cell/><Cell>Congo<Cell>4,490<Cell>3,270
26    <Row><Cell><Cell>Limpopo<Cell>2,327<Cell>1,511
27    </Row>
28    <!-- Draw red line by filling 1.5 pt body row -->
29    <Row h=1.5pt fill=0 color=Red wp=Y>
30    <Row><Cell>South America<Cell>Amazon<Cell>6,565<Cell>4,077
31    <Row><Cell><Cell>Orinoco<Cell>2,603<Cell>1,601
32    <Row><Cell><Cell>Parana<Cell>2,588<Cell>1,845
33    <Row h=1.5pt fill=0 color=Red wp=Y>
34    </Tbl>
```

Example 2.4 Rivers of the World: using <TblColFormat ... />

The output from Example 2.4 is shown in Figure 2.7.

Long Rivers of the World

Location	Name	km	Miles
Africa	Nile	6,695	4,155
	Congo	4,490	3,270
	Limpopo	2,327	1,511
South America	Amazon	6,565	4,077
	Orinoco	2,603	1,601
	Parana	2,588	1,845

Figure 2.7 Output from Example 2.4

The following is an explication of the markup used in Example 2.4. First, note that the number of columns specified is 4 (the <Tbl ... > code 'n=4' option) but only three column widths are given. This is because widths of 'surplus' columns default to the last column width provided. Alternatively, the 'count' option may be used to repeat column widths, so for example a <TblColWidth ... /> code like this:

```
<TblColWidth count=50 W=3mm/>
```

specifies a table with 50 columns, each 3 mm wide.

Setting the <TblColFormat ... /> 'A' option to L specifies that the default alignment of *all* text is left aligned. As there is only one alignment flag, L, it is applied to all columns. If the first two columns are to be left aligned, and all other columns to be right aligned, then the following could be used:

```
<TblColFormat count=2 A=L/>
<TblColFormat A=R/>
```

The <TblColFormat ... /> 'fmt' option works in the same way as the 'A' option, in that a default paragraph can be defined on a per-column basis, for example:

```
<TblColFormat fmt=P_Continent/>
<TblColFormat fmt=P_River/>
<TblColFormat fmt=P_Length/>
```

In this case the 'fmt' option sets default paragraph formats to be used for each Header, Body and Footer row, until the next <TblColFormat ... /> code is encountered.

Continuing in Example 2.4, the 'cs=4' option specified on the first <Cell ... > code in the

first heading row specifies that the contents of that cell will span 4 columns of the table.

The 'Rivers' table format (line 1 in Example 2.4) specifies that the separator between the heading and body rows should be a 1.5 point rule, colored red, and the ruling between heading rows should be a 0.3 point black rule. To get the most precise text placement within rows, especially relative to thick rules, the height of the row may be specified using the <Row ... > 'h' option.

Another method to avoid thick rules 'crowding' text is to fill a borderless row. This is illustrated by the row immediately following the entry for the Limpopo river:

```
<Row h=1.5pt fill=0 color=Red wp=Y>
```

This row is output as a red rule, 1.5 points thick. The 'wp=Y' option (keep this row with the previous row) ensures that the row does not get orphaned by a page break. If this red rule was produced by specifying an equivalent bottom rule on the 'Limpopo' row, instead of using a separate row, then the entries for the Limpopo and the Amazon would appear too close together.

Cell text vertical alignment

Vertical alignment of text within table cells may be determined by the <Cell ... > code 'Ca' option (see page R-84). This option can be used to align text at the top, center or bottom of a cell, as illustrated in Example 2.5 below. The vertical alignment property can be predefined either within a referenced table or paragraph format in a template file or by using a <ParaDef ... > code (see the <ParaDef ... > 'Ca' option on page R-363 of the *Miramo Reference Guide*). The output from Example 2.5 is shown in Figure 2.8.

```
1  <Tbl fmt="Format A" lr=None rr=None >
2  <TblColWidth count=3 W=1.5cm />
3  <Row h=17mm  >
4  <Cell fmt=output Ca=T >
5  Top text
6  <Cell fmt=output Ca=B >
7  Bottom text
8  <Cell fmt=output Ca=M >
9  Middle text
10 <P fmt=output >
11 More text
12 </Tbl>
```

Example 2.5 Controlling vertical alignment of text

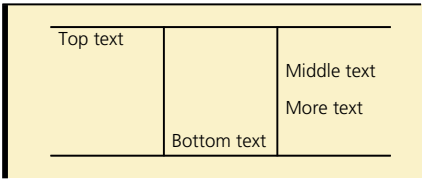


Figure 2.8 Output from Example 2.5

Note that the vertical alignment of text within table cells is also affected by the top and bottom cell margin settings. This is discussed in the section **Top and bottom cell margins and borders** on pages U-25–27.

Using 'external' data

The <?System ... ?> and <Include ... /> codes can be used within tables to incorporate externally referenced data. Example 2.6 below uses the Unix *awk* program to extract data from a Unix system file and puts the output in table columns.

```
1  <!-- Center table (A=C), -->
2  <!-- Medium rule below heading (sr=Medium), -->
3  <!-- Medium rule at top & bottom (tr & br = Medium) -->
4  <!-- Set left, top, right & bottom cell margins (clm,ctm,crm,cbm) -->
```

```
5  <Tbl fmt="Format A" A=C sr=Medium tr=Medium
6    br=Medium clm=2mm ctm=1mm crm=1mm cbm=.6mm >
7    <TblColWidth W=1.7cm /><TblColWidth W=2.1cm />
8
9  <!-- Table heading section  -->
10   <TblColFormat fmt=headex A=C />
11   <TblColFormat fmt=headex A=L/>
12   <Row type=H >
13     <Cell>Group ID</Cell>
14     <Cell>Group name</Cell>
15   </Row>
16
17 <!-- 1st table body section -->
18   <TblColFormat fmt=mtablehead A=C />
19   <TblColFormat fmt=body A=L />
20   <?System #--- Unix command
21     head -4 /etc/group ¦ awk -F: '{
22     printf("<Row><Cell>%s<Cell>%s\n", $3,$1)}' ?>
23
24 <!-- Could end the table here with </Tbl> -->
25   <Row h=1.5mm >                <!-- Blank separator row -->
26
27 <!-- 2nd table body section -->
28   <TblColFormat fmt=body A=L /><TblColFormat A=R />
29   <Row><Cell>1<Cell>daemon
30   <Row><Cell>2<Cell>kmem
31 </Tbl>
```

Example 2.6 Additional table markup

Group ID	Group name
0	root
1	other
2	bin
3	sys
1	daemon
2	kmem

Figure 2.9 Output from Example 2.6

The output from Example 2.6 is shown in Figure 2.9. The output from the *awk* command within the <?System ... ?> code in Example 2.6 (lines 21 and 22) is shown in Figure 2.7.

```
<Row><Cell>0<Cell>root
<Row><Cell>1<Cell>other
<Row><Cell>2<Cell>bin
<Row><Cell>3<Cell>sys
```

Example 2.7 Output from *awk* command

More sophisticated applications may use <?System ... ?> and <Include ... /> codes in tables to extract data directly from corporate database management systems. If the output table structure is very simple, then this can provide a quicker route to table production than embedding codes in a database report writer.

Dataless tables: chess board examples

Dataless tables in Miramo can be useful. The next example uses a standard FrameMaker default table format, 'Format A', and shows the beginnings of a chess board:

```
1  <Tbl fmt="Format A" TP=N lr=Medium tr=Medium rr=Medium br=Medium wo=20 >
2  <TblColWidth count=6 W=4mm />
3
4  <Row h=4mm >    <!--  Height of row fixed at 4mm -->
5  <Cell color=White fill=0 /> <Cell color=Black fill=0 />
6  <Cell color=White fill=0 /> <Cell color=Black fill=0 />
7  <Cell color=White fill=0 /> <Cell color=Black fill=0 />
8  <Row h=4mm>
9  <Cell color=Black fill=0 /> <Cell color=White fill=0 />
10 <Cell color=Black fill=0 /> <Cell color=White fill=0 />
11 <Cell color=Black fill=0 /> <Cell color=White fill=0 />
12
13 <!-- Above two rows Repeated below ... -->
14
15 </Tbl>
```

Example 2.8 Chess board example (1)

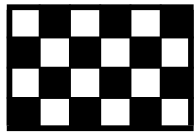


Figure 2.10 Output from Example 2.8

The table shown in Example 2.8 uses the default FrameMaker table format 'Format A'. Best programming and document management practice requires that a library of suitable table formats is included either within a template file, and is accessed using the <Template ... /> code, or by using <TblDef ... > codes, or both. However in this case <Tbl ... > options have been used to illustrate their convenience in special cases.[1]

Overriding table format defaults

The entry for <Tbl ... > in the *Miramo Reference Guide* (see pages R-251–260) contains over thirty options that can be used to override the default table format definition. Example 2.8 uses just a few of these. 'A=C' requests the table to be centered, rather than left adjusted as is normal for 'Format A' tables. 'Format A' specifies table left, top, bottom and right rulings of 'Thin'. 'Thin' is itself a default FrameMaker ruling. In the above example these rulings are overridden by 'Medium', a thicker FrameMaker default table ruling option.

While the output from Example 2.8 is a passable representation of part of a chess board, a critical eye may complain that the black squares are larger than the white squares. The source of this problem is that the 'Format A' table specification specifies that columns and rows are separated by thin black lines. These lines grow into their adjacent cells. This problem may be solved in four different ways.

- The ruling options on cells can be set to `None`, as follows:

  ```
  <Cell color="White" fill="0" lr="None" tr="None" rr="None" br="None" >
  ```

  However the output in Figure 2.11, created by setting all the cell ruling options on the third cells in rows one and two to `None`, shows that this needs to be done with care.

  (Individual cell rulings at a table border override the table border rulings.)

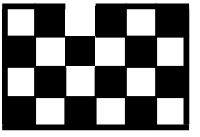


Figure 2.11 Chess board example (2)

- The format of 'Format A' in the template file or in the <TblDef ... > may be redefined to contain no inter-row or inter-column rulings.
- A new table format, 'Chess boards', may be defined by copying 'Format A' and adjusting the copy to have all the features required. Generally this is the ideal solution, since no <Tbl ... > options will then be required.
- Finally, it is possible to use a trick. This trick involves giving the table format the

1. Using <Tbl ... > options has the same general disadvantages, and in some situations, advantages, as using options on most other inline markup codes. Apart from problems associated with document management and consistency of document structure and appearance, the main disadvantage is that using <Tbl ... > options inhibits the ability of users to control the appearance of the printed output by making on-screen modifications to a template file. (However in certain cases this may exactly the effect the programmer needs to achieve.)

<Tbl ... > options are used throughout the examples shown in this chapter simply because they are an efficient way of showing how to achieve special effects. In practice the programmer is likely to want to employ the equivalent options into table format definitions, using <TblDef ... > format definition codes, or by defining the table format in a template file.

name of a non-existent table format:

```
<Tbl fmt="None" A="C" lr="Medium" tr="Medium" rr="Medium" br="Medium" >
```

This is a lazy solution. The table format name, "None", used here is arbitrary. Any name may be used provided no table format of the same name exists. A non-existent table format 'contains' no rules, however it is left aligned by default. Hence the 'A' option is set to C above.

Using any of the above solutions results in a cleaner, less 'black' chessboard.

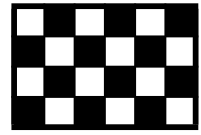


Figure 2.12 Chess board example (3)

'Dataless' tables have other potential uses: the simple color charts shown in Appendix 2 in the *Miramo Reference Guide* are produced using tables.

As a general rule using either of the Miramo pre-processors, *mmpp* or *mmxslt*, is best for producing highly repetitive output formats. As an illustration of using *mmpp*, the text in Example 2.9 could be put in a file called 'checkers.mmp':

```
1   <@macro> chessboard {
2           <Tbl fmt=None>
3           <TblColWidth count=$2 W=$3mm />
4
5           @for(a = 1 to $1) {                         @--- $1 is no. of rows --
6                   <Row h=$3mm>
7                   @for(b = 1 to $2) {                 @--- $2 is no. of cols --
8                           @if(($$b + $$a) % 2) {      @--- 1 = True --
9                                   <Cell color=Black fill=0 />
10                          }
11                          @else {
12                                  <Cell color=White fill=0 />
13                          }
14                  }
15                  </Row>
16          }
17          </Tbl>
18  }
```

Then create a file called 'checkers' containing:

```
<@include> checkers.mmp
<¦chessboard> 8 ¦ 8 ¦ 3
<¦chessboard> 12 ¦ 12 ¦ 3
```

Example 2.9 File 'checkers.mmp'

and run the command:

```
miramo -M -P checkers
```

the output will be a single printed page containing two checkerboards, the first 8 × 8 and the second 12 × 12 squares.

Note that when producing large quantities of similarly patterned output, it will be more efficient to use *mmpp* to produce mmDraw graphic objects, encapsulated within <mmDraw> and </mmDraw> codes.

Complex table

There are few limits to the complexity of tables that may be created using Miramo. The

table markup shown in Example 2.10 is only moderately complex, however it illustrates techniques which can be used to create much more complex formats including multiple row and column straddles.

```
<Tbl fmt="Format A" sr=Thick clm=0 A=R wo=20 ><!-- Thick rule under heading -->
<TblColWidth count=6 W=12.0mm />

<TblColFormat fmt=toutput A=C />

<!-- Table heading -->
<!-- First heading row -->
<Row type=H h=6mm fmt=boutput A=C >
<!-- Span 3 rows (vertical), rotate & black fill -->
<Cell rs = 3 a = 270 fill = 0 fmt=Route_Id >42
<!-- Span five columns -->
<Cell cs = 5 fmt=boutput A=C >Main heading spanning sub heads

<!-- Second heading row -->
<!-- Set row height to be exactly 5 mm -->
<Row type=H h=5mm>
<!-- Span next two rows & rotate -->
<Cell rs = 2 a = 270 crm=0>
Qty<br/>(boxes)
<!-- Span three columns -->
<Cell cs = 3>minor spanning head
<Cell rs = 2 lr = Thick A=C>Cost

<!-- Third heading row: two ways to get similar effect -->
<Row type=H h=5mm A="C">
<Cell fmt=output>head
<Cell fmt=output>head
<Cell>
<P fmt=output>
head

<!--  Table body -->
<Row h=4.25mm />            <!-- Compensate for thick rule above -->
<Row h=4mm ><Row h=4mm >
</Tbl>
```

Example 2.10 Table with spans and rotated text

The output from Example 2.10 is shown in Figure 2.13.

42	Main heading spanning sub heads				
	Qty (boxes)	minor spanning head			Cost
		head	head	head	

Figure 2.13 Table with spans and rotated text (output from Example 2.10)

Note that table dimensions can be fixed precisely using Miramo. Neither column widths nor fixed row heights are adjusted to accommodate very thick inter-column and inter-row rules or to accommodate table cell or paragraph cell margins. This makes it easy to understand what output to expect, and to make the appropriate adjustments to get desired effects. In many cases fixing row heights, possibly apart from specifying minimum row height, must be avoided in order that cells will adjust vertically to accommodate an indeterminate amount of text.

Table cell margins and table cell borders

There are several ways of controlling the size of left and right margins around text in table cells. The table codes <Tbl ... > and <TblDef ... > and the <Cell ... > code, all have cell left, top, right and bottom margin, or 'clm', 'ctm', 'crm' and 'cbm' options, and a global 'cm' option which sets all margins to the given value.

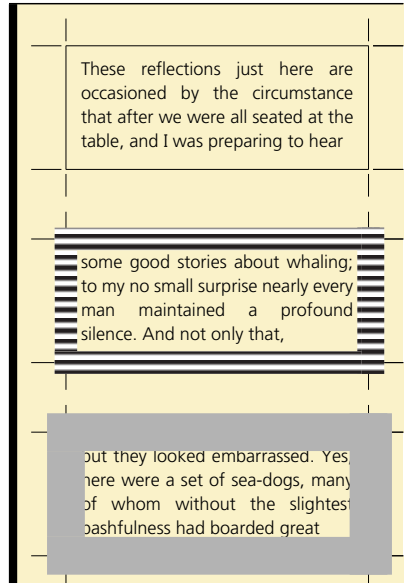


Figure 2.14 Cell margins and rulings

Cell margin values always default to zero relative to the default table cell margin values, and for this reason, and other reasons, it is most convenient to control the size of cell margins using table code cell margin option values. In the unusual cases where specific table cells must have larger or smaller left or right margins than are standard for the remainder of the table, this can be achieved using some or all of the 'clm', 'ctm', 'crm', 'cbm' or 'cm' on the <Cell ... > code.

Adjusting the left or right margin size on a per cell basis, using the 'clm', and/or 'crm' options is normally necessary only to *reduce* a particular cell's left or right margin. *Increases* in the effective margin size can be done on a per paragraph basis using the paragraph left and right indent options, 'fi', 'li' and 'ri', also supported on the <Cell ... > code for the for default margins for all paragraphs in a cell.

Cell rulings thicken about their center lines. Thickening cell borders are illustrated in Figure 2.14, using two non-solid border rulings.

Notice that as the cell borders, or rulings, thicken the position of the enclosed text does not change. This is because cell margins are always calculated from the centerlines of the cell borders. Cell borders which are wider or higher than adjoining cells will 'obliterate' the adjoining cell contents. Borders applied to cells generally override borders applied to preceding cells. Borders applied to cells at table edges will override table left, top, right and bottom borders, as well as table heading under-rules and table footer over-rules.

Top and bottom cell margins and borders

The sizes of top and bottom table cell margins may be controlled in the same way as left and right table cell margins. The only difference is that negative values of top and bottom cell margins are effective and may be used to push text outside the walls of a table cell, if desired.

Fruit	**Mangoes**	**Coconuts**	**Melons**

Figure 2.15 Text set to top of table cells ('Ca' option set to T)

In special cases a problem may arise when text of different sizes is used in the headings of multi-column tables, and the text is set to align with the tops of the table cells. This problem occurs because text is

located at the top of a table cell so that the top edge of the character bounding rectangle[1] aligns with the bottom edge of the top cell margin.

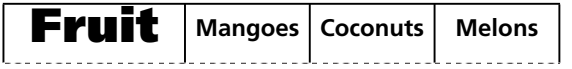


Figure 2.16 Text set to middle of table cells ('Ca' option set to M)

This is not generally what is wanted. Better output results from setting the paragraph table cell alignment property to 'Middle' instead of 'Top' (using either the Paragraph Designer in a template file or the 'Ca' option with the <Cell ... > code), *viz*.

This is effective in cases when column headings can contain from one to several lines. However when column headings each contain just one line it is often required that all the headings share the same baseline. To do this a custom top table cell margin must be set for each heading cell paragraph.

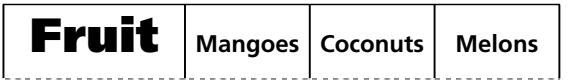


Figure 2.17 Text aligned along baselines

For example, assume that it is required to align on a baseline 18 points (a quarter inch) from the top of the cell. In this case a custom top cell margin must be set on each column heading paragraph equal to 18 points *minus* two-thirds the point size of the paragraph text. This will force synchronization of the first baselines across the column headings, as shown below:

The procedure can be generalized by defining a macro thus:

```
@-------------------------------------------------
<@macro> CellTopAlign {
    <#def>      text        {$1}        @--- Data
    <#def>      pgf         $2          @--- Paragraph suffix
    <#def>      psize       $3          @--- Point size of text
    <#def>      boff        $4          @--- Baseline offset (in points)
    <#def>      pctm        @eval(<#boff> - <#psize> * 2/3)
    @- - - - - - - - - - - - - - - - - -
    <Cell fmt=P_Cell<#pgf> p=<#psize>pt ctm=<#pctm>pt
$1
}
@-------------------------------------------------
```

Example 2.11 Utility macro for positioning text in table cells

Then only this input is needed to get aligned baselines in the column headings:

```
<Row>
<¦CellTopAlign> Fruit¦1¦18¦18
<¦CellTopAlign> Mangoes¦2¦9¦18
<¦CellTopAlign> Coconuts¦3¦9¦18
<¦CellTopAlign> Melons¦4¦9¦18
```

Unusual, but sometimes useful, output results if the above input were to be changed as follows:

```
<Row >
<¦CellTopAlign> <Font fcolor=Olive fw=UltraBlack >Trees</Font>¦1¦18¦18
<¦CellTopAlign> Baobab¦2¦9¦4
```

1. Character bounding rectangles are discussed in the chapter entitled 'Text formatting' (see Chapter 4, page U-61).

```
<¦CellTopAlign> Banyan¦3¦9¦0
<¦CellTopAlign> Beech¦4¦9¦-8
```

Example 2.12 Upshifting table cell margins

Aligning text at the bottom of table cells can be achieved in exactly the same way, except that *one third* the pointsize of the paragraph text must be subtracted from the value of the desired baseline

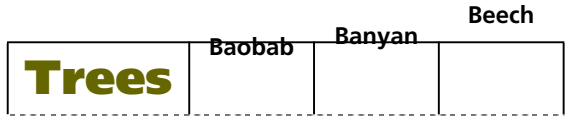


Figure 2.18 Upshifting table cell margins: output from Example 2.12

Using 'mmpp' to produce tables

The examples so far in this chapter are clear and unfortunate proof that the markup required for producing tables can be very complex.

This section shows how to use *mmpp*, the Miramo macro pre-processor, to simplify the presentation of data. This is illustrated using Example 2.10 (see pages U-24–24) and altering it to look like a timetable.[1]

The first step is to define a macro containing the table header format, along with the table body setup specification, as shown in Example 2.13.

```
<@xmacro> - sailingSchedule {          @- Precede macro name with ' - ' to store content until
                                       @- element terminated (see page R-501)
        @--- Assign default values
        <#define> tth.routeno          {0}
        <#define> tth.periodstart      {No start date}
        <#define> tth.periodend        {No end date}
        <#define> tth.deptport         {No dept port}
        <#define> tth.destport         {No dest port}
        <#define> tth.callport1        {No call port}
        <#define> tth.callport2        {No call port}
        @xgetvals(tth.)                @--- Get attribute values
        @--- Table heading
        <Tbl fmt="T_TimeTable" TP="N" sr="ThickMaroon" wo="50" cm="0" A="C" >
        <TblColWidth count=6 W=15mm />

        <TblColFormat count="6" fmt="toutput" A="C" />
        <!-- First heading row -->
        <Row type="H" h="6mm" >
        <Cell rs="3" a="270" fill="0" color="Maroon" cm="0" ctm="3.0mm" Ca="T" >
        <P fmt="Route_Id" p="32" ><#tth.routeno>
        <Cell cs="5">
        From <Font p="11"><#tth.deptport></Font> to <Font p="11"><#tth.destport>
        </Cell>

        <!-- Second heading row -->
        <Row type=H h=5mm>
        <Cell rs ="2" a ="270" p="7" >From<br/><Font fmt="F_Bold"><#tth.periodstart></
                Font><br/>to<br/><Font fmt="F_Bold"><#tth.periodend></Cell>
        <Cell cs = "3">Departure times</Cell>
        <Cell lr = "ThickMaroon">Arrival</Cell></Row>

        <!-- Third heading row -->
        <Row type="H" h="5mm">
        <Cell><Font fmt="F_Bold"><#tth.deptport>
```

1. A complete description of *mmpp* is given in the Miramo Reference Guide, Chapter 6, pages R-463–529.

```
34          <Cell><Font fmt="F_Bold"><#tth.callport1>
35          <Cell><Font fmt="F_Bold"><#tth.callport2>
36          <Cell lr = "ThickMaroon" ><Font fmt="F_Bold"><#tth.destport>
37
38          <TblColFormat count="2" fmt="toutput" A="L" />
39          <TblColFormat A="C" />
40          <Row h="2mm" wp="Y" br="None">
41          <Cell lr="None" >
42  }{
43          @--- Macro termination
44          @--- Include everything between <sailingSchedule> and </sailingSchedule>:
45          $sailingSchedule
46          </Tbl>                          @--- Table termination
47  }
```

Example 2.13 Macro definition for timetable

Example 2.13 begins with a set of definitions which store default attribute values (lines 4 through 10).

The next step is to define a macro for the table body rows, as shown in Example 2.14.

```
1   <@xmacro> sailTime {
2       <#def> st.dayName          {Daily}
3       <#def> st.times            {}
4       @xgetvals(st.)
5       <Row fmt="toutput" h="4mm" wn="Y" Ca="B" A="C">
6       <Cell lr="None" br="None" />
7       <Cell fi="1mm" A="L" ><#st.dayName></Cell>
8       @slice(<#st.times>, bsv) timeVals
9       </Row>
10  }
```

Example 2.14 Macro definition for timetable body rows

Example 2.14 includes the *mmpp* @slice() function (line 8) to parse the vertical bar-separated time values of the 'times' attribute. The @slice() function invokes a macro 'timeVals' that specifies how to process the vertical bar-separated arguments. The 'timeVals' macro is shown in Example 2.15.

```
<@macro> timeVals {
    @for(time = 1 to $# ) {
        <Cell>$time</Cell>
        }
}
```

Example 2.15 'timeVals' macro definition

The simple markup shown in Example 2.16 may now be used to create the table.

```
<!-- TIMETABLE 28  -->
<sailingSchedule   routeno="28" periodstart="Aug 3" periodend="Sep 28" deptport="Athens"
                   destport="Naxos" callport1="Tilos" callport2="Rhodos" >
<sailTime     dayName="Monday" times="06.15¦14.22 ¦20.45¦22.30" />
<sailTime     dayName="Tuesday" times="07.05¦14.40¦21.05¦22.00" />
<sailTime     times="05.35¦12.15¦20.15¦21.10" dayName="Thursday" />
<sailTime     dayName="Friday" times="06.15¦14.22¦20.45¦22.30" />
</sailingSchedule>

<!-- TIMETABLE 35 -->
<sailingSchedule   routeno="35" periodstart="Jul 15" periodend="Sep 29" deptport="Athens"
                   destport="Paros" callport1="Helos" callport2="Samos" >
<sailTime     times="06.15¦14.22¦18.45¦19.30" dayName="Sunday" />
<sailTime     dayName="Monday" times="06.15¦14.22¦18.45¦19.30" />
<sailTime     dayName="Tuesday" times="07.05¦14.40¦21.05¦22.00" />
<sailTime     dayName="Thursday" times="05.35¦12.15¦20.15¦21.10" />
```

```
<sailTime     dayName="Friday" times="06.15¦14.22¦20.45¦22.30" />
</sailingSchedule>
```

Example 2.16 Input for 'sailingSchedule'

To produce two timetables:

28 From Athens to Naxos

From Aug 3 to Sep 28	Departure times			Arrival
	Athens	**Tilos**	**Rhodos**	**Naxos**
Monday	06.15	14.22	20.45	22.30
Tuesday	07.05	14.40	21.05	22.00
Thursday	05.35	12.15	20.15	21.10
Friday	06.15	14.22	20.45	22.30

35 From Athens to Paros

From Jul 15 to Sep 29	Departure times			Arrival
	Athens	**Helos**	**Samos**	**Paros**
Sunday	06.15	14.22	18.45	19.30
Monday	06.15	14.22	18.45	19.30
Tuesday	07.05	14.40	21.05	22.00
Thursday	05.35	12.15	20.15	21.10
Friday	06.15	14.22	20.45	22.30

Either the '-M' or the '-Mfile' Miramo command line option must be used when the input stream includes *mmpp* macro calls.

Snug-fitting anchored frames in table cells

Anchored frames can be made to fit precisely within the borders of a table cell by setting the anchored frame position to 'Run into Paragraph', i.e. by setting the <AFrame ... > 'P' option to `R` (see page R-46), and by setting all the cell margins to zero (`0`, see line 10 in Example 2.17 below).

```
1  <RuleDef fmt=Wide1 pen=3 pw=0.1in color=Maroon />
2  <RuleDef fmt=Wide2 pen=3 pw=0.2in color=Maroon />
3  <Tbl fmt=None cr=Wide1 xcr=Wide2 xcn=2 TP=N >
4   <TblColWidth W= 0.1in />
5   <TblColWidth W= 1.0in />
6   <TblColWidth W= 0.15in />
7       <Row h=0.1in >
8       <Row tr=Wide1 br=Wide1 >
9           <Cell>
10          <Cell cm=0pt >
11              <AFrame P=R H=0.5in pen=0 pw=0.05in color=Blue
12              >
13              </AFrame>
14      <Row h=0.1in >
15 </Tbl>
```

Example 2.17 Snug-fitting anchored frames

The output from Example 2.17 is shown in Figure 2.19. In Figure 2.19 the outside edge of the blue bordered anchored frame is precisely aligned with the enclosing table cell border (anchored frame borders thicken from the outside inwards). If the height of the anchored frame is increased, then the height of the row will increase to accommodate the anchored frame. (This assumes that this anchored frame is the highest object in the row.)

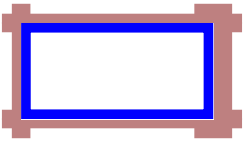


Figure 2.19 Output from Example 2.17 above (with an opaque fill on the anchored frame)

If the <AFrame ... > code on line 11 in Example 2.17 is given a 'fill' setting of 15 (transparent), as shown below:

```
11    <AFrame P=R H=0.5in pen=0 pw=0.05in color=Blue fill=15
```

then the output is as shown in Figure 2.20.

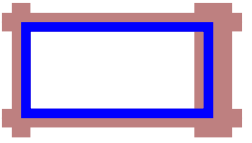


Figure 2.20 Output from Example 2.17 above (with a transparent fill on the anchored frame)

Note that the anchored frame overprints the table cell border rulings. This problem can be overcome by increasing each table cell margin value by an amount equal to half the border thicknesses and increasing the column width by an amount equal to half the sum of the left and right border widths. This is illustrated by changing line 5 in Example 2.17 is changed to:

```
5     <TblColWidth W=1.15in />
```

and changing line 10 to:

```
10    <Cell clm=0.05in ctm=0.05in crm=0.1in cbm=0.05in >
```

These changes produce the output shown in Figure 2.21.

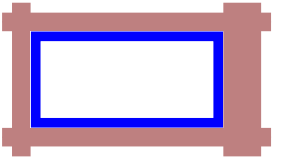


Figure 2.21 Output from Example 2.17 (modified as shown in lines 5 and 10 above)

Tables at top of columns

Tables often need to be placed at the top of a column at the beginning of a document. A common application is when the section heading text must be 'reversed' out of, or over-printed on, a solid background color. If this section heading text naturally breaks so that it is placed at the top of the next column in the flow, it must be aligned correctly at the top of the column.

This can be done by anchoring such tables within an otherwise empty paragraph, and set this paragraph's 'space below' and the table's 'space above' to have negative values equal to the point size of the paragraph. The actual space above the table can be varied by changing the 'space above' value on the enclosing paragraph. This space above will get 'lost' when the paragraph occurs at the top of a page.

```
<P fmt=Body p=2pt sb=-2pt wp=N>
<Tbl fmt=SectionHead sa=-2pt>
<TblColWidth W=2.8in /> <!--  Width of column  is 2.8 inches -->
<Row fill=0 color=Black>
<Cell fmt=SectionHead color=White>
Section Title Text
</Cell>
```

```
</Row>
</Tbl>
```

Example 2.18 Section heading within a table

Reversing text out of, or printing text on top of, a solid background requires sensitive alignment of top, bottom and side margins. This alignment is achieved either by using the <Tbl ... > or <Cell ... > 'clm', 'ctm', 'crm' and 'cbm' options or by adjusting the equivalent values using the Table Designer or the Paragraph Designer in a template file.

Tables within tables

The <TblFrame ... > code enables tables to be shrink-wrapped within anchored frames. This makes it possible to have 'run-in' tables, tables located outside the main text flow or tables located in any of the positions supported by the <AFrame ... > code.

The <TblFrame ... > code also enables tables to be placed within tables. Such tables within tables may be nested to any level. Example 2.19 below shows how this can be done.

The discussion and examples in this section relate only to container tables and included tables that have fixed-width columns. For tables and included tables that have proportional-width columns see section **Auto-sizing tables to width of table cells** on pages R-273–275 in the *Miramo Reference Guide*.

To reduce unnecessary processing overhead avoid using <TblFrame ... > codes unless any of the following are required: tables within tables, images behind tables, color gradient or partly transparent fills in cells, rows or entire tables (using the <FrameFill ... /> code described on pages R-148–158 in the *Miramo Reference Guide*), or tables located in places which can only be reached using the <AFrame ... > code (see the <AFrame ... > 'P' option on page R-46 in the *Miramo Reference Guide*).

The output from Example 2.19, shown in Figure 2.22 on page U-32, is annotated with the <P ... > code, <TblFrame ... > code, <TblColWidth ... /> code and <TblTitle ... > code options which affect the total size and extent of the <TblFrame ... > area. These options are shown in bold in the Example 2.19 code listing (lines 10 through 16).

Example 2.19 contains three tables. The first table, which starts on line 2, is a simple instance of a <TblFrame ... >. This contains a second table, which starts on line 13, in the second column of the second row of the first table. The second table itself contains a third, three-column table (which starts on line 32) straddling columns one and two of the fourth row in the second table. The third table contains the letters A, B and C, one in each column.

```
1   <!-- Table 1 ---------- -->
2   <TblFrame fmt="Format A" sb="0" sa="0" P="R" A="R" fRAgap="10pt" >
3   <TblColWidth W= "18pt"/><TblColWidth W="148pt"/>
4   <TblTitle fmt="P_TblTitle" p="12pt" Tgap="2pt" >
5   Title: 1st table</TblTitle>
6   <Row h=3mm/>
7   <Row>
8   <Cell>
9   X
10  <Cell fill="5" color="Black" fmt="Body" clm="18pt" ctm="14pt" cbm="12pt" >
11          <!-- Table 2 ---------- -->
12          <TblFrame fmt="Format A" n="3" P="R" sa="16pt" sb="10pt" li="16pt" ri="18pt"
```

```
13              ffill="0" fcolor="LightBrown" fill="15" >
14          <TblColWidth count="2" W="24pt" />
15          <TblColWidth W="40pt" />
16          <TblTitle fmt="P_TblTitle" p="10pt" Tgap="14pt" >Title: 2nd table</TblTitle>
17          <TblColFormat fmt="P_OutputCell" />
18          <Row >
19              <Cell><Font fmt="F_Bold">Fish</Cell>
20              <Cell><Font fmt="F_Bold">Birds</Cell>
21              <Cell><Font fmt="F_Bold">Mammals</Cell>
22          </Row>
23          <Row >
24              <Cell>Shark<Cell>Eagle<Cell>Tiger
25          </Row>
26          <Row >
27              <Cell>Cod<Cell>Parrot<Cell>Camel
28          </Row>
29          <Row a="0" lr="Thin" rr="Thin" >
30              <Cell cs="2" clm="0" ctm="0" crm="0" cbm="0">
31              <!-- Table 3 ---------- -->
32              <TblFrame fmt="Abcd" TP="N" P="R" A="L" fpen="0" fpw="2pt" >
33              <TblColWidth count="3" W="16pt" />
34              <TblColFormat fmt="P_OutputCell" A="C" />
35              <Row lr="Thin" rr="Thin" >
36                  <Cell><P>A</P></Cell>
37                  <Cell><P>B</P></Cell>
38                  <Cell><P>C</P></Cell>
39              </Row>
40              </TblFrame>
41              </Cell>
42          <Cell>and others</Cell>
43          </Row>
44          </TblFrame>
45  </TblFrame>
```

Example 2.19 Tables within tables

The output from Example 2.19 is shown in Figure 2.22.

In the following discussion the 'tableframe area' is the area enclosed by the anchored frame containing the '2nd table' and is shown in Figure 2.22 with a light brown background, surrounded by a dark red border. The line numbers refer to the code listing shown in Example 2.19, which begins on page U-31.

The width of the table frame area is the sum of 'li' and 'ri' (see line 13) plus the sum of the widths of the three columns in the table, that is 16 + 18 + 24 + 24 + 40 (122 points). The height of the table frame area is the sum of 'sa', the height of the table title,[1] the height of the 'gap' between the table title and top of the table (see line 16), the sum of all the row heights in the table and 'sb'. (The values of 'sa' and 'sb' are set on line 13.)

If the <TblFrame ... > 'P' option is set to R (i.e. the enclosing anchored frame has the 'Run into paragraph' property set), the top of the table frame area is offset from the top of the second cell in the second row of the

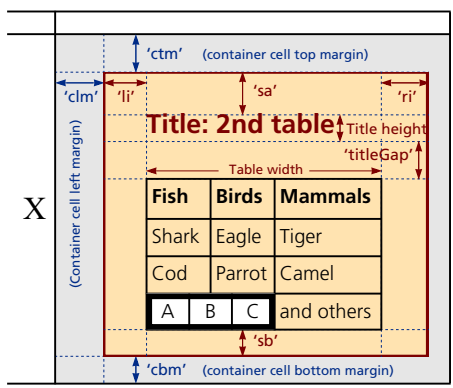


Figure 2.22 Tables within tables (output from Example 2.19 above)

first table either by the default table cell top margin or, as here, the absolute value of the 'ctm' option supplied to the first paragraph in the cell (see line 10 — the cell margin sizes are always controlled either by the default settings for the table or by the settings on the <Cell ... > code). Similarly the bottom of the second row of the first table is offset from the bottom of the table frame area by the value of the cell bottom margin, 'cbm' (line 10 again). The cell margin area is shown in gray (see line 2). The value of the cell right margin of the cell containing the table frame is immaterial to the positioning (and width, of course) of the table frame.

If the sum of 'clm' and the total width of the table frame area is greater than the total width of the column containing the table frame, then the right hand side of the table frame will be cropped.

To achieve precise control over the positioning of the table frame area when no surrounding text is included in the enclosing table cell always set the <TblFrame ... > 'P' option to `R`. This way the font size and location options of the first paragraph in the enclosing cell are immaterial.

The background color of the table frame area is set using the <TblFrame ... > 'ffill' and 'fcolor' options. The default background color of *all* the cells in a table instance may be set using the 'fill' and 'color' options. Fill patterns and colors set on individual cells, using the same options, will override the default for the table as a whole. To set the table background color to be the same as the background color of the table frame area it is safest always to set the <TblFrame ... > 'fill' option to `15`.

The combination of options on the <TblFrame ... > code allows many alternative routes to achieving relatively simple results.

Title: 1st table

	Fish	Birds	Mammals
X	Shark	Eagle	Tiger
	Cod	Parrot	Camel
	A B C		and others

Figure 2.23 Tables within tables (2)

Assuming a table cell without surrounding text is required then complexity may be reduced first by setting all the enclosing cell margins to zero, i.e. set the values of 'clm', 'ctm' and 'cbm' to 0 (line 10 in Example 2.19 on page U-31) and, second, by removing the table title. Since a table frame can never be longer than a single page having a title is rarely useful: the title text can be placed in the first table row. Thirdly the <TblFrame ... > 'li', 'ri', 'sa' and 'sb' may all be set to zero (`0`). Figure 2.23 shows the effect of setting all the foregoing parameters to zero.

This simplification produces good results provided either the enclosed table has the same outside rulings as the enclosing cell or both have no rulings. However if neither of these conditions is true then it is necessary to set one or more of the foregoing 'li', 'sa', 'ri' and 'sb' values to allow space for these rulings. If these values are not set the rulings will be cropped to half their thicknesses, since table rulings thicken equally inwards and outwards from cell borders.

1. In the simplest case, a table title consisting of text all in the default paragraph font size, the title height is the same as the point size of the text. If the table title consists of *n* lines of text, then the title height is *n* × 'p' *plus* 'l' × (*n* - 1). 'p' is the text point size and 'l' is the inter-line leading. Further adjustments are needed if the title contains larger-than-default character size settings.

Images in table frames

The <FrameImage ... /> code enables images to be placed behind the table frame area. The image is automatically sized to fit exactly within the borders of the table frame area, as illustrated in Example 2.20 below. There is no way to preserve the original aspect ratio of such images.

```
1  <TblFrame n=4 fmt=T_ColorGrad P=b sa=0 sb=0 A=R >
2    <TblColWidth W=36mm />
3    <TblColWidth W=18mm />
4    <TblColFormat fmt=P_OutputCell A=L/>
5    <TblColFormat A=L />
6    <TblColFormat A=L />
7    <TblColFormat A=R />
8  <!-- Background image -->
9  <FrameImage file="${IMAGES}/goldgrad.eps" />
10 <Row type=H>
11   <Cell><Font fmt="F_Bold">Description
12   <Cell><Font fmt="F_Bold">Period
13   <Cell><Font fmt="F_Bold">Av. Qty
14   <Cell><Font fmt="F_Bold">$ per gm
15 <Row>
16   <Cell>Black diamonds<Cell>Jan - Mar<Cell>248<Cell>452.83
17 <Row tr=WhiteB>
18   <Cell>Coramandel rubies<Cell>Feb - Jul<Cell>139<Cell>321.02
19 <Row tr=WhiteB>
20   <Cell>Klondike gold 20s<Cell>Sep - Dec<Cell>286<Cell>123.47
21 <Row tr=WhiteB>
22   <Cell>Shanghai blues<Cell>Jan - May<Cell>315<Cell>263.12
23 </TblFrame>
```

Example 2.20 Table with color gradient

The output from Example 2.20 is shown in Figure 2.24. The file 'goldgrad.eps', referred to in line 3 of Example 2.20, is a simple color gradient saved in EPS format using Adobe Illustrator.

Description	Period	Av. Qty	$ per gm
Black diamonds	Jan - Mar	248	452.83
Coramandel rubies	Feb - Jul	139	321.02
Klondike gold 20s	Sep - Dec	286	123.47
Shanghai blues	Jan - May	315	263.12

Figure 2.24 Output from Example 2.20 above

Producing a multi-page table with a color gradient requires that each table header, body and footer row contains a <TblFrame ... > code. The color gradient image and the data must be contained within this <TblFrame ... > code. The enclosing table should comprise just one column, wide enough to contain all the <TblFrame ... > column data. Note that the formatting time for such tables will be much longer than for a simple table.

An alternative way of achieving gradient fills, with optional transparency, is to use the <FrameFill ... /> code, described on pages R-148–158 in the *Miramo Reference Guide*.

Combining text, tables and images within table cells

Many directories and industrial catalogs require images to be placed in table cells and also

to have runaround text in the same cell. Normally this is achievable simply by having an <AFrame ... > in the table cell. Example 2.21 shows a more complicated implementation containing a background color gradient and an image with a table super-imposed upon it.

```
<Tbl fmt="Format A" >
<TblColWidth W=18pt /><TblColWidth W=8.0cm />
<Row>
  <Cell lr=None Ca=M color=Maroon>X</Cell>
  <Cell rr=None cm=0 >
       <TblFrame fmt="Format A" P=R sa=0mm sb=0mm clm=0 ctm=0 cbm=0 >
       <TblColWidth W=8cm />
       <FrameImage file=${IMAGES}/goldgrad.eps />
       <Row>
       <Cell rr=None clm=2mm ctm=2mm cbm=1mm crm=3mm>
       <P fmt=P_Input >
This is some run a round text.  This is some run a round text.
              <TblFrame fmt=T_ColorGrad P=R >
              <TblColWidth W=24pt />
              <TblColWidth W=26pt/>
              <TblColWidth W=40pt/>
              <TblColFormat fmt=P_OutputCellWhite />
              <FrameImage file=${IMAGES}/030112_inv.jpg />
              <Row >
                <Cell fw=Bold >Fish
                <Cell fw=Bold >Birds
                <Cell fw=Bold >Mammals
              <Row >
                <Cell>Shark<Cell>Eagle<Cell>Tiger
              <Row >
                <Cell>Cod<Cell>Parrot<Cell>Camel
              <Row>
                <Cell cs=2><Cell>and others
              </TblFrame>
       <P fmt=P_Input >
This is some more run a round text.  This is some run a round text.
This is some more run a round text.  This is some run a round text.
       </TblFrame>
</Tbl>
```

Example 2.21 Text, tables and images in table cells

The output from Example 2.21 is shown in Figure 2.25.

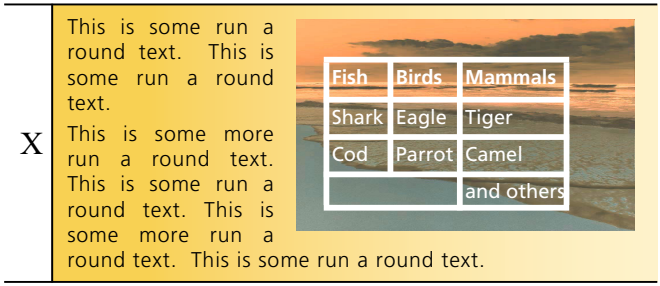


Figure 2.25 Text, tables and images in table cells

CHAPTER 3

Anchored frames and images

Anchored frames are rectangular boxes which can be placed within a text column, or in margin space adjacent to it, or in table cells. These frames are 'anchored' in a paragraph, or in a table cell, using the <AFrame ... > code. The full set of options for the <AFrame ... > code is described in the *Miramo Reference Guide* on pages R-45–55.

There are several <AFrame ... > options to control the location of anchored frames. These options include settings to display at the top or bottom of the current page, inline within the text precisely at the anchor point, immediately below the line containing the anchor point, in the left or right margins, or in the outer or gutter margins. Several frames may be anchored to the same point.

The <Image ... /> code is used to place vector or raster images within anchored frames (as well as within unanchored frames, master pages and background text columns). The <Image ... /> code supports several location and orientation options which are detailed in the *Miramo Reference Guide* on pages R-180–190. Using the <AFrame ... > code and the <Image ... /> code is the key to including graphics, images and fixed frames within document text flows.

The <FrameImage ... /> code (see pages R-159–160 in the *Miramo Reference Guide*) is used to apply background images to tables created using the <TblFrame ... > code and to anchored text frames (see the <ATextFrame ... > code).

Supported graphic image file formats

The fully supported image file formats are listed (alphabetically) below. Supported image formats can be imported using the 'file' option on the <Image ... /> code (see page R-180).

- **BMP** [Win NT only]

  BMP is a Windows bitmap image format that comprises images with 1, 4, 8, 16, 24 or 32 bits per pixel.

- **CGM**

  CGM (Computer Graphics Metafile, ISO 8632-1–4) is a multi-level, multi-encoding combined vector and bitmap graphics format common in military and aerospace environments.

- **EPS** (Encapsulated PostScript)

  Encapsulated Postscript files are specially structured PostScript programs describing a single page, or part of a page. EPS files usually contain a preview bitmap image for on-screen display, which may be binary or ASCII (EPSI) encoded. The PostScript program within an EPS file may include a PostScript image in binary or ASCII form, or may comprise PostScript text and drawing commands, or a combination of both. EPS files may be included in bypass mode (see the <Image ... />

code 'importMode' option on page R-180) in the *Miramo Reference Guide*.

- **GIF**

  GIF (Graphics Interchange Format) files are primarily used on the Internet. GIF uses a lossless, LZW based compression scheme and supports a maximum of 256 colors (1 to 8 bits per pixel).

- **JPEG** (and **JFIF**)

  The JPEG (Joint Photographic Experts Group) lossy compressed image format is particularly suited to storing contone, or photographic, images. JPEG comprises several different compression modes. Miramo supports the most common mode, baseline Huffman encoding.

  A JFIF (JPEG File Interchange Format) file is a JPEG file containing a special JFIF data record. A JFIF data record may contain data describing additional image attributes, such as the width and height of the image in absolute units (inches or centimeters). This additional data in a JFIF file is irrelevant to, and ignored by, the import process.

  JPEG images may be included in bypass mode (see the <Image ... /> code 'importMode' option on page R-180) in the *Miramo Reference Guide*. This means that if the <Image ... /> code 'importMode' option is set to `bypass` all versions of JPEG format images supported by the installed version of Acrobat Distiller may be imported and displayed in PDF output.

  If a JPEG (or JFIF) file contains more than one image, only the first image is imported. JPEG files do not support transparency.

- **PNTG (MacPaint)**

  A MacPaint file comprises a single bitmapped, fixed-size image 576 pixels wide, 720 pixels high and 1-bit deep. The bitmap data is compressed using PackBits encoding.

- **PCX**

  The PCX bitmap image file format was developed by ZSoft for use with their PC Paintbrush drawing and painting program. It supports monochrome images and 8-bit and 24-bit color images.

- **PNG**

  The PNG (Portable Network Graphics) bitmap image file format, developed by the PNG Development Group partly, provides lossless support for up to 48 bits per pixel.

  PNG images may be included in bypass mode (see the <Image ... /> code 'importMode' option on page R-180) in the *Miramo Reference Guide*. Full support for alpha channel transparency in PNG files is provided when the <Image ... /> code 'importMode' option is set to `bypass`.

- **SNRF (Sun raster)**

  Sun raster files may be uncompressed or run length encoding (RLE) compressed.

They may be 1-bit monochrome images, 8-bit color images containing a color map or 24-bit RGB images.

- **TIFF**

  The Tagged Image File Format (TIFF) format is a 'suitcase' format capable of storing a wide range of bitmap image encodings, compression schemes, color maps and color models. Files conforming to Baseline TIFF, as defined in the TIFF Revision 6.0 specification can be imported.

  TIFF files which use the CCITT G4 and LZW compression formats, defined as part of the 'Extensions' included in the TIFF Revision 6.0 specification can also be imported.

  TIFF images may be included in bypass mode (see the <Image ... /> code 'importMode' option on page R-180) in the *Miramo Reference Guide*. Support for alpha channel transparency in TIFF files is provided when the <Image ... /> code 'importMode' option is set to `bypass`.

- **xwd**

  X-Windows Dump (xwd) file format files are the standard output of the *xwd* X-Windows screen capture program.

Many applications have interpreted the standards, or documents, describing the above supported formats in different ways. This means that in some cases a file purporting to be in one of the supported formats may not be importable.[1]

Importing 'un-supported' image file formats

In addition to the supported file formats listed above, several 'partially supported' formats can be imported, for example: CITT G3, CITT G4, DXF, DRW, GEM, QuickDraw PICT and WPG. Importing partially supported image file formats requires that the <Image ... /> code always has the 'W' and 'H' (width and height) options set, since the aspect ratio of such images is not checked.

The following discussion of importing images always relates to supported image types, unless otherwise noted.

<Image ... /> code options

All images are imported using the <Image ... /> code (see pages R-180–190). One or more <Image ... /> codes may occur within an <AFrame ... >, <Frame ... >, <PageDef ... > and <TextFrameDef ... > (i.e. background text frames only) code pairs. Unlike the foregoing codes, which must always be followed by a matching termination code, the <Image ... /> code is a content-free code.

Imported images may be scaled, flipped, rotated, masked and surrounded by borders, as desired. Scaling, either maintaining a constant aspect ratio, or by varying X and Y

1. In this situation first try checking the file using the *itell* utility (type `itell -help`) and, secondly, try importing and re-exporting the image file in another application such as PhotoShop or Adobe Illustrator.

amounts, is achieved using the 'H' *or* 'W' *or* 'dpi', or using the 'H' *and* 'W', <Image ... /> options. The 'dpi' option has the lowest priority, i.e. it is ignored if either the 'H' or 'W' <Image ... /> options are used.

All options work in the same way on all *supported* image file formats, with the exception of EPS files. As explained in the examples given in the section entitled **Encapsulated PostScript files** later in this chapter (page U-44), the <Image ... /> code 'dpi' option cannot be used with EPS files.

Simple examples

All the following examples using PCX files, apply equally to any of the supported image file formats, and will produce equivalent results.

The image shown in Figure 3.1 was created using Windows Paintbrush. Its dimensions as reported by Paintbrush are 2.513 cm wide and 1.984 cm high at 96 dpi.

Figure 3.1 Image centered within an anchored frame

The anchored frame and image shown in Figure 3.1 is created using the input shown in Example 3.1.

```
<AFrame W="29.13mm" H="23.84mm" A="C" P="b" pen="0" pw="1mm">
    <Image file="${IMAGES}/stairway.pcx" dpi="96" pen="0" pw="0.5" L=".2cm" T="2mm" />
</AFrame>
```

Example 3.1 Image centred within an anchored frame

In Example 3.1 the anchored frame's width and height are specified to be 29.13 mm and 23.84 mm using the <AFrame ... > 'W' and 'H' options, with a 1 mm frame ruling set to 100% black (pw=".1cm", pen="0"). The frame anchor point is set to be below the point where the <AFrame ... > code is included (P="b"), and the frame alignment is centered within the text column (A="C"). Setting the <Image ... /> 'dpi' option to `96` scales the included image to 100%, and the image is indented 2 mm from the top and left sides of the enclosing anchored frame (T="2mm", L="2mm"). A black, 0.5 point border is drawn around the image (pen="0", pw="0.5"). The units of the 'pw' option always default to points unless the value is followed by the units name (as in the first line of Example 3.1).

If the pen ('pen') and pen width ('pw') options are omitted from both the <AFrame ... > and <Image ... /> codes, as shown in Example 3.2)

```
<AFrame W=29.135mm H=23.84mm A=C >
    <Image file="${IMAGES}/stairway.pcx" dpi=96 L=2mm T=2mm  />
</AFrame>
```

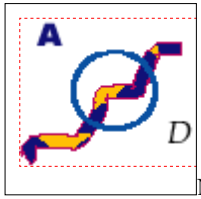
Example 3.2 Borderless image

then the output is as shown in Figure 3.2.

Figure 3.2
Same as Figure 3.1 above, without frame or image borders

(The dashed line and the red dotted line in Figure 3.2 above are position indicators for the anchored frame and image borders. They are not printed or displayed.)

Removing all options except the <AFrame ... > pen options, the <Image ... /> filename (always required) and the displacement options, and adding 'P=l',



produces an inline image and enclosing frame: Note that to avoid obscuring the text above inline frames, the line spacing of the enclosing paragraph must not be set to 'Fixed' in the 'Basic Properties' of the paragraph format menu of the template file. Alternatively, if the paragraph format is specified within the <ParaDef ... > statement within the Miramo input file, then the 'ls' option must be set to `P` or `A` within this specification.

'Below current line' positioning ('P' option) of anchored frames is the default, i.e. if no alternative 'P' value is set Miramo assumes the anchor point setting 'P=b'.

Note that the enclosing frame is now a square, and the circle in the image has become an ellipse. This is because if either the width or height option is omitted from the <AFrame ... > or <Image ... /> inline code, the omitted dimension defaults to a value of 25.4 mm (1"), unless the <AFrame ... > 'wrap' option is set to `Y` (see page R-45 in the *Miramo Reference Guide*.

Image scaling, rotation and cropping

The six diagrams below, along with the following markup code illustrate image scaling, rotation and cropping.

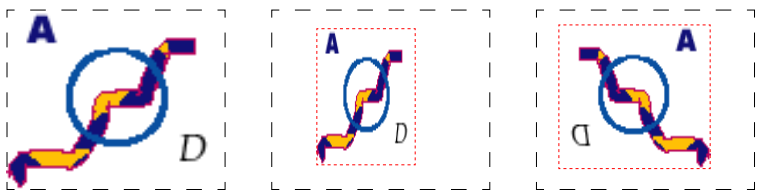


Fig. 3.3 (1) Fig. 3.3 (2) Fig. 3.3 (3)

Figure 3.3 (1) - (6) Image transformations

Fig. 3.3 (4)

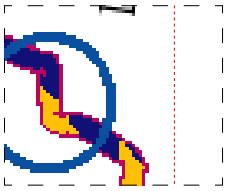


Fig. 3.3 (5)

Fig. 3.3 (6)

Figure 3.3 (1) - (6) Image transformations (*Continued*)

The code sections below are used to produce the examples in Figure 3.3. The labels ahead of each code section refer to the to the figures within Figure 3.3.

3.3 (1)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${MMAN}/images/stairway.pcx" W=28.9mm H=23.8mm />
</AFrame>
```

3.3 (2)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${IMAGES}/stairway.pcx" W=13mm H=18mm L=6mm T=2.5mm />
</AFrame>
```

3.3 (3)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${IMG}/stairway.pcx" flip=Y W=20mm H=19mm L=3mm T=2mm />
</AFrame>
```

3.3 (4)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${IMG}/stairway.pcx" W=40mm H=32.5mm T=-7mm L=-10mm />
</AFrame>
```

3.3 (5)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${IMG}/stairway.pcx" W=4cm H=3.25cm T=-7mm L=-1cm a=90 />
</AFrame>
```

3.3 (6)
```
<AFrame W=28.9mm H=23.8mm A=C P=b pen=0 pw=0.1 >
<Image file="${IMG}/stairway.pcx" W=31.1mm H=25mm /> <!-- scale by 1.25 >
</AFrame>
```

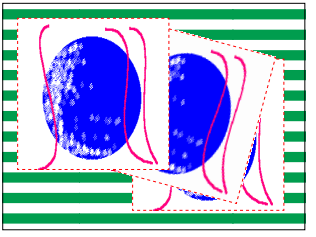
Multiple images in a single frame

Several images may be included within a single <AFrame ... >. This can be achieved as in Example 3.3.

```
<AFrame H=30mm W=40mm A=C P=b pen=0 pw=0.15 A=R P=R RAgap=5mm >
<Image file="${IMG}/bars.eps" H=30mm W=40mm />
<Image file="${IMG}/circle.pcx" H="20mm" W="20mm" L="17.2mm" T="7.4mm" />
<Image file="${IMG}/circle.pcx" H="20mm" W="20mm" L="11.6mm" T="2mm" a="345" />
<Image file="${IMG}/circle.pcx" H="20mm" W="20mm" L="2mm" T="2mm" />
</AFrame>
```

Example 3.3 Multiple images within an anchored frame

Normally images will be placed adjacent to each other. However images may be superimposed one upon another without limit. In this case later <Image ... /> codes will place their images on top of images referenced in preceding <Image ... /> codes. Setting the <Frame ... > code 'P' option to R causes the output from Example 3.3 to be shown at right.



JPEG images

JPEG (Joint Photographic Experts Group) encoding is an efficient way of compressing contone images, e.g. high quality color photographs scanned at a high resolution. Miramo imports JPEG files in just the same way as it imports PCX files.

The main drawback of JPEG files is their high processing overhead. For this reason using large JPEG files in environments where very high-speed output is required should be avoided.

The next examples are different presentations of the same scanned image.

Use:

```
<AFrame W=3.6cm H=3.9cm A=C pen=0 pw=0.5pt color=Maroon >
    <Image file="${IMG}/scales.eps" W=36mm H=39mm />
    <Image file="${IMG}/030113a.jpg" dpi=600 />
</AFrame>
```

Example 3.4 Using the 'dpi' option

to print the image at its original aspect ratio and scaled at 600 pixels to the inch. The output from Example 3.4 is shown in Figure 3.4 at right.

Figure 3.4 Using the 'dpi' option

Alternatively, use:

```
<AFrame W=3.5cm H=2.9cm A=L P=R >
    <Image file="${IMAGES}/030113a.jpg" W=3.5cm H=2.9cm />
</AFrame>
```

Example 3.5 Absolute scaling, with distortion

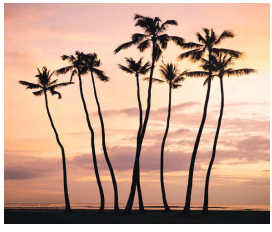


Figure 3.5 Output from Example 3.5

to scale the image to fixed dimensions on both the x-axis and the y-axis:

The following:

```
<AFrame W=60.8mm H=36mm A=C P=R A=R>
    <Image file="${IMG}/030113a.jpg"
        W=16.24cm H=24cm L=-61.6mm T=-30.4mm />
</AFrame>
```

Example 3.6 Zooming-in and cropping an image

will produce an enlarged and cropped version of the image:

Figure 3.6 Zooming-in and cropping an image (Output from Example 3.6)

TIFF files

TIFF was developed in 1986 when Aldus Corporation and several scanner vendors developed a standard file format for images used in desktop publishing. Version 5 TIFF was published in 1988. Version 6 TIFF was published in 1992.[1]

TIFF supports a wide variety of bitmap image types: everything from binary monochrome, to grayscale, to full 24-bit color. Several different types of compression encoding schemes are supported. These vary depending on the bitmap image type.

Miramo can reference all types of Baseline TIFF files. Baseline TIFF formats comprise:

- Bilevel or monochrome images, with either no compression, CCITT Group 3 one dimensional compression or PackBits compression.
- Grayscale images comprising 4 or 8 bits per pixel, with either no compression or PackBits compression.
- Palette-color images comprising 4 or 8 bits per pixel, with either no compression or PackBits compression.
- RGB full color images comprising 24 bits per pixel, 8 bits for each of the RGB colors in contiguous format, with either no compression or PackBits compression.

Miramo can also reference TIFF files that incorporate two compression formats defined as part of the TIFF extensions, *viz.* CCITT G4 fax encoding[2] and LZW compression.

Encapsulated PostScript files

As a general rule, highest quality printed output is achievable using encapsulated PostScript files. When such files contain only PostScript code, and no bitmap image data, they are scalable without any loss in image quality.

The encapsulated PostScript (EPS) file format is a 'standard format' for importing and exporting PostScript language files in a variety of heterogeneous environments.[3] An

1. The only reliable source of information about the TIFF format is contained in the *TIFF Revision 6.0 Specification* itself, originally published by Aldus Corporation on June 3rd 1992. This document is now available from: http://partners.adobe.com/asn/developer/technotes.html

2. Sometimes referred to as 'T6-encoding', with reference to CCITT T.6 bi-level encoding as specified in section 2 of CCITT Recommendation T.6: *Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus*, International Telephone and Telegraph Consultative Committee (CCITT, Geneva, 1988).

encapsulated PostScript file is a PostScript language program describing the appearance of a single page (at most) or part of a page. An EPS file can contain any combination of text, vector graphics and bitmap images.

The acronyms and nomenclature associated with EPS files are often misused. They include: 'EPS', 'EPSF', 'EPSI' and 'EPS binary', all of which may refer to different types of encapsulated PostScript files. The confusion is worsened since all encapsulated PostScript files, regardless of type, must contain the string 'EPSF' as a part of their identifying header. The differences between encapsulated PostScript files all relate to the format of the screen preview image included in the file. This usually varies depending on the platform and the application which created the EPS file.

There are five different types of EPS file. Two of these types relate to Windows non-device independent formats.

- The term 'EPS' may be used to denote all or any type of encapsulated PostScript file. However it is perfectly valid for an encapsulated PostScript file to have no preview image included within it, and sometimes the term 'EPS file' is used to denote such an encapsulated PostScript file.

  The Miramo distribution binaries for Unix include the 'fmps2eps' utility which will generate such an EPS file from a single page FrameMaker 6 or 7 PostScript file.

- A file referred to as an 'EPSF' file is usually created on a Macintosh. In this case it contains a preview image in PICT format. (The Macintosh file type for application-created PostScript files is EPSF.) Beware though that 'EPSF' is sometimes used as a generic term for all types of encapsulated PostScript files.

  A problem with using Macintosh EPSF files is that often only part of the 'file' gets transported to other platforms. Usually the Resource fork, containing the preview image is left behind. This means that though EPSF files may be referenced by Miramo and will print OK, no image will display on screen.

- On Windows systems, files ending with '.EPS' are nearly always encapsulated PostScript files with a preview image in either Windows Metafile or TIFF format. In addition to the binary preview image, which is often located at the end of the encapsulated PostScript file, these files always begin with thirty bytes (octets) of binary header data which include the byte offset and extent of the preview image. The first four bytes in the file must be hexadecimal C5D0D3C6 (byte 0 = C5). These files are sometimes referred to as 'EPS binary'.

- 'EPSI' files are encapsulated PostScript files with a preview image in a device independent format. The preview section of an EPSI file is in ASCII hexadecimal for simplicity and ease of transport (though not for space economy).

The nearest approximation to an authoritative definition of these different types of encapsulated PostScript files is contained in *PostScript Language Reference Manual, Second Edition* (see footnote reference above).

---

3. See *PostScript Language Reference Manual, Second Edition*, Adobe Systems Incorporated, Addison-Wesley, Reading, Mass. 1990. ISBN 0-201-18127-4, Appendix H, pages 709 - 736.

Miramo can reference EPSI, EPSF and 'EPS binary', files as well as encapsulated PostScript files with no image preview data. Every EPS file which is to be used with Miramo must start with a line which begins:

```
%!PS-Adobe-2.0 EPSF-2.0
```

or

```
%!PS-Adobe-2.0 EPSF-2.0
```

or either of the above preceded by 30 binary bytes (octets, see above).

Every EPS file must contain '%%BoundingBox' data as follows:

%%BoundingBox: *llx lly urx ury*

This line generally occurs, and strictly speaking should occur, anywhere within the EPS file header comments section, before the '%%EndComments' line. It is essential that this line is present before the '%%BeginProlog' comment. Occasionally an application will place the %%BoundingBox data at the end of an EPS file. In this exceptional case the above line may be replaced with a line of the form

```
%%BoundingBox: (atend)
```

and another %%BoundingBox comment with appropriate *llx lly urx ury* arguments will be found at the end of the file, in the %%Trailer section.

The %%BoundingBox parameters *llx lly urx ury* specify the co-ordinates of the lower left and upper right corners of a rectangle enclosing the image. Ideally this rectangle will be the *smallest* rectangle into which the whole image will fit. Unfortunately in many cases it will be the 'page size' selected when making the drawing. The units are always points (72 points = 1 inch).

A small segment from the beginning of a typical EPSI file with a device-independent screen display format is shown in Example 3.7.

```
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: -1 -21.03 200 300
%%Creator: Drawing program
%%Title: triangle.eps
%%CreationDate: Fri Jul  2 09:00:28 1999

%%DocumentFonts:
%%EndComments
%%BeginPreview: 75 75 1 75
%00000000000000000000
%00000000000000080000
```

Example 3.7 EPSI image file: sample of initial data

Manually editing the values of the %%BoundingBox parameters will change how the EPS file is displayed on screen, but will have no effect on printed output. Changing both the displayed image and printed image may be output by using the Miramo <AFrame ... > (or <Frame ... >) and <Image ... /> codes. These markup codes enable EPS files to be enlarged, reduced or cropped in a consistent fashion.

Though the Adobe Document Structuring Conventions (DSC) specify that %%Bounding-Box parameters must be integer values, this is often ignored by applications, as in the

above example. Alternatively, some applications, for example Adobe Illustrator, include a %%HiResBoundingBox as shown below:

```
%%HiResBoundingBox: 29.58 217.98 582.3 573.9
```

If the EPS file contains such data, it will be used by Miramo.[1]

By default, Encapsulated PostScript files are referenced by, not copied into, the Miramo file. Adding the <Image ... /> code 'imode=C' option to the examples below will cause these image files to be copied into the Miramo output file.[2]

The following markup code produces the image shown below:

```
<AFrame H=3.4cm W=5.4cm A=C pen=0 pw=0.15 >
    <Image file="${IMG}/tri.eps" H=3cm W=2.4cm pen=0 pw=0.05 L=.2cm T=.2cm />
    <Image file="${IMG}/tri.eps" H=3cm W=2.4cm L=2.8cm T=.2cm flip=Y a=180
        imode=C  />
</AFrame>
```

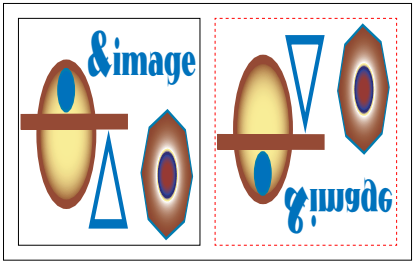


Figure 3.7 Distorted, inverted and flipped EPS images

This code:

```
<AFrame H=3.4cm W=5.4cm A=C pen=0 pw=0.15 >
    <Image file="${MMAN}/images/tri.eps" H=7cm W=8cm L=-1cm T=-1cm />
</AFrame>
```

produces the output shown in Figure 3.8 at right.

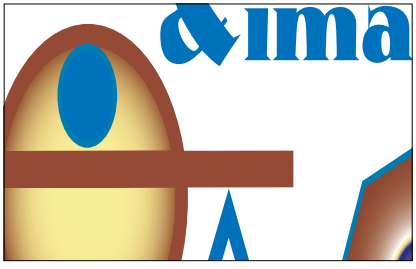


Figure 3.8 Masked EPS image

In all the above examples the EPS image is scaled to the dimensions specified by the 'W' and 'H' <Image ... /> options. The next example shows the EPS image height scaled to equal the specified width divided by the aspect ratio:

```
<AFrame H=3.3cm W=2.3in A=C pen=0 pw=0.15 >
    <Image file="${IMG}/tri.eps" W=2in L=.2cm T=.2cm  />
</AFrame>
```

1. This may be checked using the Miramo *itell* utility.

2. Copying an image file into a Miramo output file is an advantage only if it is required to transport the Miramo output file to a different environment to that in which it was created. An example of this is if the final output files are to be displayed on networked PCs using FrameReader or FrameViewer.

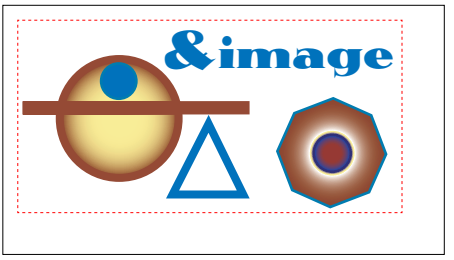


Figure 3.9 Automatic scaling — using the <Image ... /> code 'W' option only

The next example shows the EPS image width scaled to equal the specified height multiplied by the aspect ratio:

```
<AFrame H=3.4cm W=5.4cm A=C pen=0 pw=0.15 >
    <Image file="${IMG}/tri.eps" H=30mm L=1mm T=2mm  />
</AFrame>
```

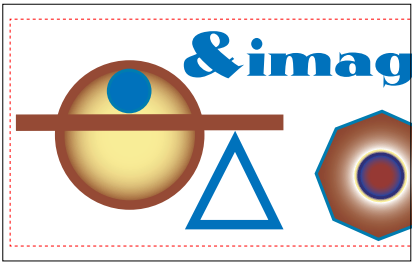


Figure 3.10 Automatic scaling — using the <Image ... /> 'H' option only

The %%BoundingBox and %%HiResBoundingBox DSC comments included in the EPS file containing this image are:

```
%%BoundingBox: 168 484 313 557
%%HiResBoundingBox: 168.495 484.6985 312.505 556.8333
```

i.e. the width is 144.01 points (50.804 mm) and the height is 72.1348 points (25.45 mm). When an EPS image contains a %%HiResBoundingBox statement it is used in place of the %%BoundingBox statement.

Fixed anchored frame & image widths

Sometimes images and their enclosing anchored frames must be fitted to a constant width. This may be the case for example in component catalogs, where each image may be required to fit within the width of a table cell, the height of the table row containing the image automatically adjusting to the image height or to the height of another cell in the row, if that is greater.

Constant-width images can be output by shrink wrapping an <Image ... /> code, with its 'W' option set to the required constant value, within an <AFrame ... > code having the 'wrap' option set to Y, a illustrated in lines 8 through 10 in Example 3.8.

```
1  @--------------------------------------------------------------------
2  <@xmacro> fixedWidthImage {
3        <#def> fwi.width      0.5in       @--- Default image width
4        <#def> fwi.file       {Dummy}     @--- Default file name
5        @xgetvals(fwi.)       @- Get attribute vals for width & file, if any
6        <#def> fwi.file       {${IMAGES}/<#fwi.file>}
7        <P fmt="P_Body" sa="0" sb="0" p="2">
```

```
8         <AFrame wrap="Y" pen="0" pw="0.5" >
9             <Image file="@trim(<#fwi.file>)" W="<#fwi.width>" />
10        </AFrame>
11    }{}
12    @-------------------------------------------------------------------
13    <fixedWidthImage file="100024.tif" />
14    @-------------------------------------------------------------------
```

Example 3.8 Scaling images and anchored frames to a fixed width (1)

In Example 3.8 the markup to enable shrink wrapping anchored frames around images scaled to a constant width is encapsulated in a *mmpp* X macro (see pages R-500–515 in the *Miramo Reference Guide*). The effect is to create a new markup code, <fixedWidthImage/>, that has optional attributes (see line 13 in Example 3.8). This technique requires that the '-M' or '-Mfile' command line option is used (see page R-5 in the *Miramo Reference Guide*).

A 1 inch × 1 inch empty frame is output and a warning message is issued if the <Image ... /> 'file' option is set to a file name of an image that is corrupted or does not exist. A more sophisticated version of the *mmpp* macro in Example 3.8 protects against such errors. This macro, shown in Example 3.9, skips problematic images and writes error data to a log file.

```
1   @-------------------------------------------------------------------
2   <@xmacro> fixedWidthImage {
3         <#def> fwi.width      0.5in       @--- Default fixed image width
4         <#def> fwi.filename   {dummy}     @--- Default image file name
5         @xgetvals(fwi.)
6         <#def> fwi.filename   {${IMAGES}/<#fwi.filename>}
7         @if(@image(@trim(<#fwi.filename>))) {          @--- Check image is OK
8               <P fmt="P_Body" sa="0" sb="0" p="2">
9               <AFrame wrap="Y" pen="0" pw="0.5pt" >
10              <Image file="@trim(<#fwi.filename>)" W="<#fwi.width>" />
11              </AFrame>
12              }
13        @else {
14              <@write> images.log {Missing image file: <#fwi.filename>}
15              }
16  }{}
17  @-------------------------------------------------------------------
18  <fixedWidthImage filename="100024.tif" width="0.5in" />
19  <fixedWidthImage filename="bb0004-02.gif" width="0.5in" />
20  @-------------------------------------------------------------------
```

Example 3.9 Scaling images and anchored frames to a fixed width (2)

Note the use of the *mmpp* @image( ) function within an @if( ) expression on line 7 in Example 3.9. @image( ) returns a value of 1 (success) or 0 (failure), depending on whether it succeeds in determining the required image parameters. (The *mmpp* @image( ) function is described in the *Miramo Reference Guide*, on pages R-521–522.)

The @image( ) function may only be used within an expression — it cannot be used like this:

```
<#def>  img_ok      @image(filename)          @--- ERROR! ---
```

The equivalent, correct, usage is:

```
<#def>  img_ok      @eval(@image(filename))   @--- Correct ---
```

If @image( ) succeeds in determining the image file parameters, it sets the values of the

special *mmpp* variables <#img.ar>, <#img.dpi>, <#img.h>, <#img.w>, <#img.type> and <#img.err>. The value of <#img.err> is zero on success and otherwise a number in the range 1 to 5, indicating the type of error. The value of <#img.dpi> is always zero when the file type is EPS.

The images output by Example 3.9 are shown in the rightmost column in Figure 3.11.

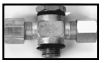
	File	W (in)	H (in)	W/H	Image
Fig 3.11 (1)	100024.tif	0.500	0.291	1.719	
Fig 3.11 (2)	bb0004-02.gif	0.500	1.520	0.329	

Figure 3.11 (1) - (2) Fixed width images

The *mmpp* Xmacro 'fixedWidthImage' in Example 3.9 (shown on page U-49) will never cause an error, since all its action is dependent on the @image( ) function succeeding in returning a '0' (success) value.

However if the image file is missing or is in a bad format the 'fixedWidthImage' macro will simply output nothing, with results that may be unfortunate. In practical applications it is recommended to insert error checking code to handle bad or missing images. Example 3.12 in the next section includes one possible solution to this problem in the form of a utility *mmpp* macro called 'image.fail'. This utility macro, or a more sophisticated variation, could be used to amend Example 3.9.

Fitting images within maximum height & width limits

An alternative requirement to scaling images to a constant width, without control of the resulting image height, is fitting images and their enclosing anchored frames within a maximum permitted width and a maximum permitted height. In this case images will be scaled to the maximum width unless the resulting height is greater than the specified maximum height 0.7 inches in Example 3.10.

Fitting images within maximum height and width limits is only a little more complicated than forcing all images to have the same width. The 'imageMaxWH' macro in Example 3.10 shows one way in which it can be done.

```
1  @--------------------------------------------------------------------
2  <@xmacro> imageMaxWH {
3       <#def> imwh.file          {Dummy}     @- Default image file name
4       <#def> imwh.maxWidth      {0.5in}     @- Default max width
5       <#def> imwh.maxHeight     {0.7in}     @- Default max height
6       @xgetvals(imwh.)          @- Get attribute vals: file, maxWidth, maxHeight
7       <#def> imwh.file {${IMG}/<#imwh.file>}
```

```
8       <P fmt=P_Body sa=0 sb=0 p=2>
9       <AFrame wrap="Y" pen="0" pw="0.1" A="R"  >
10          <Image file="<#imwh.file>"  W="<#imwh.maxWidth>" H="<#imwh.maxHeight>" pen="0"
                pw="0.1" maxWH="Y" />
11      </AFrame>
12  }{}
13  @-------------------------------------------------------------------
14  <imageMaxWH file="100024.tif" />
15  <imageMaxWH file="bb0004-02.gif" />
```

Example 3.10 Fitting images to a maximum width and height (1)

The major difference between Example 3.10 above and Examples 3.8 and 3.9 (on page U-49) is that the <Image ... /> code in Example 3.10 has both the 'W' and 'H' options set, *and* the qualifier option 'maxWH' set to Y. Setting 'maxWH' to Y changes the interpretation of the <Image ... /> code 'W' and 'H' options from absolute values to *maximum permitted* values.

The output from Example 3.10 is shown in Figure 3.12.

	File	W (in)	H (in)	W/H	Image
Fig 3.12 (1)	100024.tif	0.500	0.291	1.719	
Fig 3.12 (2)	bb0004-02.gif	0.230	0.700	0.329	

Figure 3.12 (1) - (2) Images fitted within a maximum height and a maximum width (1)

Example 3.11 is similar to Example 3.10 above, except it uses the image aspect ratio returned by the *mmpp* @image() function to calculate the 'W' and 'H' option values for the <AFrame ... > and <Image ... /> codes explicitly, rather than using the <Image ... /> code 'maxWH=Y' option setting to request automatic resizing for the imported image.

Example 3.11 also contains code to print information about missing or corrupt images in the formatted output.

```
1   @-------------------------------------------------------------------------
2   <@xmacro> imageMax {
3       <#def> imax.imgFolder       {${IMAGES}}   @- Default image file name
4       <#def> imax.file            {Dummy}       @- Default image file name
5       <#def> imax.maxWidth        0.5in         @- Default max image width (units)
6       <#def> imax.maxHeight       0.7in         @--- Default max image height (units)
7       @xgetvals(imax.)
8       <#def> imax.file            {<#imax.imgFolder>/<#imax.file>}
9       <P fmt=P_Body sa=0 sb=0 p=2>
10      @if(@image(<#imax.file>)) {
11          @if(<#imax.maxWidth> / <#img.ar> > <#imax.maxHeight>) {
12              <#def> img.w        @eval(<#imax.maxHeight> * <#img.ar>)
13              <#def> img.h        <#imax.maxHeight>
14              }
15          @else {
16              <#def> img.w        <#imax.maxWidth>
17              <#def> img.h        @eval(<#img.w>/<#img.ar>)
18              }
19          <AFrame W="<#imax.maxWidth>" H="<#img.h>" pen="0" pw="0.1" A="R"  >
20          <Image file="<#imax.file>"  W="<#img.w>" H="<#img.h>" pen="0" pw="0.1pt" />
```

```
21              </AFrame>
22              }
23          @else {
24              <¦image.fail> @basename({<#imax.file>}, d)
25              }
26  }
27  @-------------------------------------------------------------------
28  <imageMax file="100024.tif" />
29  <imageMax file="bb0002-02.gif" />
30  <imageMax file="nix.gif" />
31  <imageMax file="bb0004-02.gif" />
32  <imageMax file="/tmp" />
33  <imageMax file="bb0014-02.gif" />
34  <imageMax file="file.txt" />
```

Example 3.11 Fitting images to a maximum width and height (2)

Lines 3 through 6 in Example 3.11 define default values for the image file name as well as image size limits. Line 11 contains a check to determine whether the height of the image at its maximum permitted width will exceed the maximum permitted height. If this is the case, the image is narrowed just sufficiently to make its height equal the maximum permitted height. Otherwise the image width is set to the maximum permitted image width. Finally the <AFrame ... > and <Image ... /> codes are output with required 'W' and 'H' values. All the foregoing assumes that the @image( ) function successfully interprets the image.

If the @image( ) function (line 10 in Example 3.11) fails for any reason, then the 'image.fail' utility macro is called (see line 24) with the image file name as its argument. The 'image.fail' macro is shown in Example 3.12.

```
1   <@macro> image.fail {
2       <#def> u        in          @- Dimensional units
3       <AFrame W="<#max.iw><#u>" H="<#ih.fail><#u>" pen="0" fill="<#fill.f>" >
4       <TextFrameDef L="0" T="2pt" W="<#max.iw><#u>" H="<#ih.fail><#u>" fill="<#fill.f>">
5       <P fmt="Body" p="6.5" fi="2pt" li="2pt" l="1pt" ri="2pt" A="L">
6   File: $1<br/>
7               @if(<#img.err>  = 1)        {= Not found }
8               @elif(<#img.err> = 2)       {= Not<br/>readable }
9               @elif(<#img.err> = 3)       {= Folder<br/>name }
10              @elif(<#img.err> = 4)       {= Bad<br/>format? }
11              @elif(<#img.err> = 5)       {= Problem interpreting file }
12              @else                       {? Problem unknown }
13      </TextFrameDef>
14      </AFrame>
15  }
```

Example 3.12 'image.fail' error reporting macro

The output from Example 3.11 is shown in Figure 3.13.

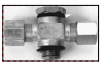
	File	W (in)	H (in)	W/H	Image
Fig 3.13 (1)	100024.tif	0.500	0.291	1.719	

Figure 3.13 (1) - (7) Images fitted within a maximum height and a maximum width (2)

	File	W (in)	H (in)	W/H	Image
Fig 3.13 (2)	bb0002-02.gif	0.464	0.700	0.663	
Fig 3.13 (3)	$1				File: nix.gif = Not found
Fig 3.13 (4)	bb0004-02.gif	0.230	0.700	0.329	
Fig 3.13 (5)	$1				File: tmp = Not found
Fig 3.13 (6)	bb0014-02.gif	0.500	0.225	2.218	
Fig 3.13 (7)	$1				File: file.txt = Bad format?

Figure 3.13 (1) - (7) Images fitted within a maximum height and a maximum width (2) (*Continued*)

Using *mmxslt* image processing functions

This section illustrates how to achieve the same effects as shown in the previous section, but using *mmxslt* in place of *mmpp*.

The XML input file in Example 3.13, 'imagelist.xml', comprises two elements. The first, <GraphicTable>, is the root element. The second, <Graphic>, is a content-free element having just one attribute, image-file, whose value is the name of a file containing an image.

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <GraphicTable>
3  <Graphic image-file="100024.tif" />
4  <Graphic image-file="bb0002-02.gif" />
5  <Graphic image-file="nix.gif" />
6  <Graphic image-file="bb0004-02.gif" />
7  <Graphic image-file="bb0014-02.gif" />
8  <Graphic image-file="file.txt" />
9  </GraphicTable>
```

Example 3.13 XML input file: 'imagelist.xml'

The structure of the 'imagelist.xml' file and its data are arbitrary. It is the equivalent of the key-coded data items shown at the end of Example 3.11.

An XSL stylesheet to process the input shown in Example 3.13 is shown in Examples 3.14 and 3.15. Note that this code is effective whether the XML input file contains just

one <Graphic> element or several thousand <Graphic> elements.

The *mmxslt* image processing extension functions are illustrated on lines 38 through 41 in Example 3.14. A description of all the *mmxslt* image processing extension functions is contained in the *Miramo Reference Guide* on pages R-420–421.

```
1  <?xml version='1.0'?>
2  <!-- Stylesheet element includes "miramo" namespace to allow   -->
3  <!-- access to miramo extension functions   -->
4  <xsl:stylesheet
5   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
6   xmlns:miramo="http://ExternalFunction.miramo.com"
7   exclude-result-prefixes="miramo" >
8
9   <!-- Set output encoding and indent   -->
10  <xsl:output encoding="US-ASCII" indent="yes"/>
11
12  <!-- Global variables   -->
13  <xsl:param name="IMG" />
14  <xsl:variable name="max.iw" select="0.5"/>          <!-- Max. image width          -->
15  <xsl:variable name="max.ih" select="0.7"/>          <!-- Max. image height         -->
16  <xsl:variable name="ih.fail" select="0.4"/>         <!-- Image `fail' height       -->
17  <xsl:include href="tblhead.xsl" />
18  <xsl:include href="imagefail.xsl" />
19  <!-- Process all Graphic elements and place output in table   -->
20  <xsl:template match="GraphicTable">
21   <P fmt="P_Tbl" >
22   <Tbl lr="None" tr="None" rr="None" br="None" cr="None" xcr="None" A="R" >
23    <!-- Include "do-tblhead" template, shown in Example 3.15 on page U-55.   -->
24    <xsl:call-template name="do-tblhead"/>
25    <xsl:apply-templates select="Graphic"/>
26   </Tbl>
27   </P>
28  </xsl:template>
29
30  <!-- Graphic   -->
31  <xsl:template match="Graphic">
32
33   <!-- FileName variable constructed from <Graphic> 'file' attribute   -->
34   <xsl:variable name="fileName" select="concat($IMG,'/',@image-file)" />
35   <Row tr="MediumMaroon" wn="N" >
36   <Cell><xsl:value-of select="@image-file"/></Cell>
37   <xsl:choose>
38    <xsl:when test="miramo:getImageStats($fileName)">
39     <Cell><xsl:value-of select="miramo:imageWidth()"/></Cell>
40     <Cell><xsl:value-of select="miramo:imageHeight()"/></Cell>
41     <Cell><xsl:value-of select="format-number(miramo:imageAR(), '#.00')"/></Cell>
42     <Cell>
43      <!-- Calculate height of anchored frame -->
44      <xsl:variable name="af.height">
45       <xsl:choose>
46        <xsl:when test="$max.iw div miramo:imageAR() > $max.ih">
47         <xsl:value-of select="$max.ih"/>
48        </xsl:when>
49        <xsl:otherwise>
50         <xsl:value-of select="$max.iw div miramo:imageAR()"/>
51        </xsl:otherwise>
52       </xsl:choose>
53      </xsl:variable>
54      <AFrame W="{$max.iw}in" H="{$af.height}in" A="R" P="R" pw=".1" pen="0" >
55       <!-- Set maximum width/height for Image -->
56       <Image file="{$fileName}" W="{$max.iw}in" H="{$max.ih}in" maxWH="Y" />
57      </AFrame>
58     </Cell>
59    </xsl:when>
60    <xsl:otherwise>
61     <!-- Include "do-imagefail", shown in Example 3.16 on page U-56.   -->
```

```
62          <xsl:call-template name="do-imagefail"/>
63        </xsl:otherwise>
64      </xsl:choose>
65      </Row>
66    </xsl:template>
67  </xsl:stylesheet>
```

Example 3.14 XSL transformation file: 'imagelist.xsl'

Note that Example 3.14 references two external files 'tblhead.xsl' and 'imagefail.xsl' (see lines 17 and 18).

The contents of 'tblhead.xsl' are shown in Example 3.15.

```
1   <?xml version='1.0'?>
2   <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3         version="1.0">
4   <xsl:template name="do-tblhead">
5         <TblColWidth W="22mm" />
6         <TblColWidth W="12mm" />
7         <TblColWidth W="12mm" />
8         <TblColWidth W="12mm" />
9         <TblColWidth W="22mm" />
10        <TblColFormat fmt="P_Output" fw="Bold" />
11        <Row type="H">
12          <Cell>File</Cell><Cell>W (in)</Cell>
13          <Cell>H (in)</Cell><Cell>W/H</Cell><Cell>Image</Cell>
14        </Row>
15        <TblColFormat fmt="P_Output" />
16  </xsl:template>
17  </xsl:stylesheet>
```

Example 3.15 Table header file: 'tblhead.xsl'

The effect of the 'call-template' on line 24 in Example 3.14 is that the code on lines 5 through 15 in Example 3.15 (i.e. the contents of template 'do-tblhead') gets included in Example 3.14 at line 24.

The contents of 'imagefail.xsl' are shown in Example 3.16.

```
1   <?xml version='1.0'?>
2   <!-- Stylesheet element includes "miramo" namespace to allow    -->
3   <!-- access to miramo extension functions    -->
4   <xsl:stylesheet
5     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
6     xmlns:miramo="http://ExternalFunction.miramo.com"
7     exclude-result-prefixes="miramo" >
8
9   <xsl:template name="do-imagefail">
10    <Cell cs="3" />
11    <Cell>
12      <AFrame W="{$max.iw}in" H="{$ih.fail}in" A="R" P="R" pen="0" fill="5">
13        <TextFrameDef L="0" T="2pt" W="{$max.iw}in" H="{$ih.fail}in" fill="5">
14        <P fmt="Body" p="6.5" fi="2pt" li="2pt" l="1pt" ri="2pt">
15  File: <xsl:value-of select="@image-file"/>
16        <xsl:choose>
17          <xsl:when test="miramo:imageErr() = 1"> = Not found</xsl:when>
18          <xsl:when test="miramo:imageErr() = 2"> = Not<br/>readable</xsl:when>
19          <xsl:when test="miramo:imageErr() = 3"> = Folder<br/>name</xsl:when>
20          <xsl:when test="miramo:imageErr() = 4"> = Bad<br/>format?</xsl:when>
21          <xsl:when test="miramo:imageErr() = 5"> = Problem interpreting file</xsl:when>
22          <xsl:otherwise> ? Problem unknown</xsl:otherwise>
23        </xsl:choose>
24        </P>
25        </TextFrameDef>
26      </AFrame>
27    </Cell>
```

```
28  </xsl:template>
29  </xsl:stylesheet>
```

Example 3.16 Image error processing: 'imagefail.xsl'

The effect of the 'call-template' on line 62 in Example 3.14 is that the code on lines 10 through 27 in Example 3.16 (i.e. the contents of template 'do-tblhead') gets included in Example 3.14 at line 62.

Running the appropriate command shown in Example 3.17 or Example 3.18

```
mmxslt -indent 5 -param IMG \"%IMG%\" -xsl imagelist.xsl imagelist.xml
```

Example 3.17 Running mmxslt using the 'imagelist.xsl' stylesheet (NT 4 & 5)

```
mmxslt -param "IMG" \"$IMG\" -xsl imagelist.xsl imagelist.xml
```

Example 3.18 Running mmxslt using the 'imagelist.xsl' stylesheet (Unix)

produces valid Miramo input. Further processing by Miramo produces the output shown in Figure 3.14.

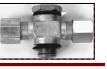
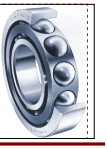
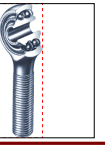
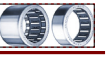
File	W (in)	H (in)	W/H	Image
100024.tif	758	441	1.72	
bb0002-02.gif	175	264	.66	
nix.gif				File: nix.gif = Not found
bb0004-02.gif	100	304	.33	
bb0014-02.gif	275	124	2.22	
file.txt				File: file.txt = Bad format?

Figure 3.14 Output from Example 3.14

CHAPTER 4

Text formatting

Paragraphs

The paragraph is a fundamental entity in Miramo. All text and graphic objects that can occur in a document must be placed within a paragraph: this includes tables, images, markers of all types, variables, etc.

Miramo provides multiple methods of controlling the layout and appearance of text in a paragraph. These methods are: using a FrameMaker template file to define paragraph and character format definitions; using <ParaDef ... > and <FontDef ... /> format definition codes; and using options with the <P ... > and <Font ... > inline markup codes. The first of these methods is discussed in the FrameMaker user documentation. The second and third methods are documented in the *Miramo Reference Guide* (see <ParaDef ... > on pages R-359–367, <FontDef ... /> on pages R-335–341, <P ... > on pages R-220–230 and <Font ... > on pages R-137–142).

There are over sixty options which control the layout and formatting of paragraph text (see the <P ... > code options on pages R-220–228 in the *Miramo Reference Guide*). The usage and effect of most of these options is simple and obvious, and only a very small subset of them are discussed in this chapter. The aim in this chapter is first to illustrate basic techniques, and second to clarify those aspects of Miramo and FrameMaker text formatting that may appear obscure, particularly as they affect vertical line positioning.

Page and column breaks

To get an instance of a paragraph to start on a new page use:

```
<P fmt="pgfname" P="P">
```

This will cause the paragraph text and/or any included objects to start at the top of the next available page. Alternatively, to get a paragraph to start at the top of a left or right page use:

```
<P fmt=pgfname P=L>
```

or

```
<P fmt=pgfname P=R>
```

Note that the 'P' option will be ineffective unless it follows the 'fmt' option. If the <P ... > code is used with no 'fmt' option, as in the following:

```
<P P="R">
```

then the 'P' option will always be effective. The following paragraph will inherit all the attributes of the preceding paragraph. For a double-sided document, if the current position is either a left page or a right page, setting the 'P' option to `L` or `R` respectively will cause a blank page to be printed, with running headers and footers, prior to text being output at the top of the next page.

To start text at the top of the next text column in the text flow (which could be on the same page, if the document is a multi-column document) use:

```
<P fmt="tagname" P="C">
```

Using <P ... > codes without options

Paragraphs beginning with <P ... > codes, without a 'fmt' option setting, inherit the defined formatting characteristics of the preceding paragraph. If the preceding <P ... > code has a 'fmt' option setting *tagname* has additional options applied to it, these options will be carried through to succeeding paragraphs without a *tagname*:

```
<P fmt="tagname" P="C">
This paragraph will have formatting characteristics defined by
<FontRef fmt=F_NoLang >&#x2018;</FontRef>tagname&#x2019;. It will start at the top of the
next column in the textflow.
<P>
This paragraph will have formatting characteristics defined by the
preceding paragraph.  It will start at the top of the next column
in the text flow.
<P fmt="tagname" >
This paragraph will start at the next available position (i.e. not
necessarily at the top of a text column).
```

<P ... > codes do not have to be terminated by </P> codes, except in the case of XML input.

Inter-word spacing

Inter-word spacing may be controlled by setting options in the FrameMaker 'Paragraph Designer' in a template file. Inter-word spacing may also be controlled using the 'gws', 'ows' and 'lws' options available for both the <ParaDef ... > and the <P ... > code. These values specify percent changes from the default inter-word spacing. A value of 100 equates to one quarter of the size of an em-space in the paragraph's default font. The 'Optimum word spacing' option (ows) takes precedence over the greatest and least word spacing options (gws and lws). In the case of left *or* right justified paragraphs setting the 'ows' option to 0 results in no inter-word spacing. If the paragraph has *both* left and right justification turned on, then positive inter-word spacing will be applied to all output lines, except for the last.

The default percent values for 'gws', 'ows' and 'lws' are 110, 100 and 90 respectively. Examples of the effect of changing these values are shown below.

```
<P fmt=Body A=L ows=100 lws=90 gws=110>
```

> For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief.

```
<P fmt=Body A=L ows=400 lws=400 gws=400>
```

> For the most wild, yet most homely narrative which I am about to pen, . . .

```
<P fmt=Body A=L ows=400 lws=0 gws=600>
```

> Mad indeed would I be to expect it, in a case where my very senses reject their

```
<P fmt=Body A=L ows=0 lws=0 gws=0>
```

> Yet,madIamnot—andverysurelyIdonotdream.Buttomorrow Idie,andtodayIwouldunburdenmysoul.

Inter-word spacing settings apply to a whole paragraph. There are no inline markup codes nor any <FontDef ... /> code options to adjust inter-word spacing for parts of a paragraph. The same effect may be achieved by constructing a composite paragraph from a series of run-in paragraphs.

Negative inter-word spacing is not supported.

Character spread

Character spread may be controlled by several different methods:

- Using the FrameMaker 'Paragraph Designer' to set character spread in a template file.
- Using the 'Character Designer' in a template file to adjust the spread value, and then either using the 'F=*fontag*' option with the <ParaDef ... > or <P ... > inline markup codes or using the '<Font fmt=*fontag*>' inline markup code.
- Using the 'Ks' option with the <P ... >, <ParaDef ... > or <FontDef ... /> codes, or the '<Font Ks=*n*>' inline markup code.

The value for inter-character spread is expressed as a percentage of an em-space in the current font. A value of 0 means that no extra inter-character spacing is added. A value of 100 means that an em-space is added between each character. Negative values reduce character spread.

'Small' variations in character spread have a large visual effect:

```
<P fmt=Body A=L Ks=0>
```

> My immediate purpose is to place before the world... a series of mere household events.

```
<P fmt=Body A=L Ks=5>
```

> I have mentioned some of the more prominent august calamities on record;

```
<P fmt=Body A=L Ks=10>
```

> but in these it is the extent, not less than the character

Character spread may be adjusted using inline markup thus:

```
<P fmt=P_Body A=L Ks=0>
which so <Font Ks=50>vividly<Font Ks=0> impresses
the fancy.
```

> which so v i v i d l y impresses the fancy.

Note that spread is applied *after* each character: to apply wider (or negative) spread *between* a set of characters in a string insert the inline mark-up required to re-adjust the spread of the string before the last character in the string.

Negative character spread enables a lot of text to be fitted into a small space, but is difficult to read:

```
<P fmt=Body A=L Ks=-8>
```

> To be buried while alive is, beyond question, the most terrific

Vertical character positioning

The standard vertical, or 'y', reference point for characters and for lines of text is the baseline: the zero 'y' value used in font metric tables is by convention co-incident with the baseline. The FrameMaker formatting engine uses two additional vertical reference points for characters and lines of text. These additional reference points are the top and bottom edges of a bounding rectangle which (usually) enclose each character. The height of the bounding rectangle is the pointsize of the font being used.[1]

The Frame formatting engine places the baseline of all characters exactly one third the way up this bounding rectangle.

1. Note that the height of FontBBox, the term used in Type 1 PostScript fonts and in PostScript font metrics files, to express the size of a rectangle capable of enclosing all the characters in a font when superimposed one upon another, is often 20% or more higher than the pointsize of the font.

Character bounding rectangles

Several characters, all having a size of 12 mm, are shown in their bounding rectangles in Figure 4.1.

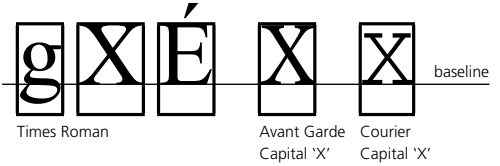


Figure 4.1 Character bounding rectangles

Most characters fit within the top and bottom of the FrameMaker character bounding rectangle. In the case of the Times font all characters fit within the bounding rectangle, except for uppercase composite characters, i.e. uppercase characters with diacritical marks. This may be seen in the case of the letter 'É' in Figure 4.1. Some fonts have a very small capital letter height, and fit into the bounding box with space to spare at the top. Courier, a 'square' font originally designed for typewriters, is an example of such a font, as illustrated in the rightmost character above. Many fonts will contain standard ascenders which stick out of the top of the bounding rectangle, e.g. AvantGarde, as shown in Figure 4.1.[1]

It is rare for a descender to extend below the bottom of the bounding box.

Text at column and table cell tops

The top and bottom edges of the FrameMaker character bounding rectangles are used as reference points for positioning characters at the top and bottom of text columns and within table cells.

Text located at the top of a text column is generally positioned with the top edge of the character bounding rectangles exactly aligned with the top of the text column. This means that the baseline of text in the first line of a paragraph at the top of a column is placed at a distance equivalent to two thirds of the text pointsize below the column top.

If any part of a character is higher than its bounding rectangle, then that part will protrude above the top of the text column, as may be seen in Figure 4.2.

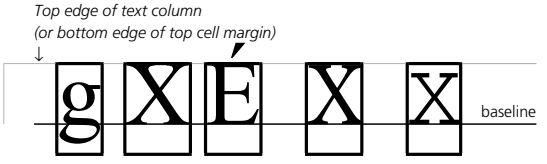


Figure 4.2 Characters at column and table cell tops

Note that the accent above the letter 'E', and the tops of the ascenders of the first letter 'X' following, protrude above the top edge of the text column. Turning 'Fixed' line spacing off makes no difference in this case.

In the case of table cells, paragraph text is positioned so that the top edge of the character bounding rectangle aligns with the bottom of the top cell margin, unless the <P ... > or <ParaDef ... > codes have the 'Ca' option set to `M` or `B`. If any part of a character is higher than its bounding rectangle, then that part will protrude into the cell margin, or if cell

1. In most PostScript fonts the baseline is set roughly 25% of the way up the FontBBox. Because FrameMaker sets the baseline 33% of the way up its bounding box a substantial number of characters grow through the top, like shoots above the rim of a flower pot. The effect is generally unnoticeable at small pointsizes.

margin is zero, into the cell above.

Text at column and table cell bottoms

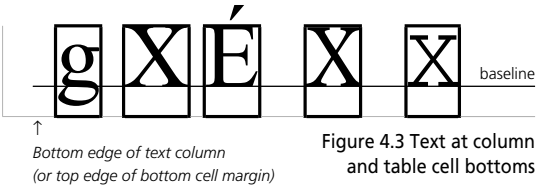


Figure 4.3 Text at column and table cell bottoms

In most cases characters will align with the bottom of a text column only if feathering is switched on. If feathering is switched on, the bottom edges of the character bounding rectangle are aligned with the bottom edge of the text column. This means that the baseline of the paragraph text is actually placed at a distance *above* the bottom edge of the text column equivalent to one third of the default paragraph point size.

Feathering is not applicable in table cells. However text will align with the top edge of the bottom cell margin if the cell vertical alignment of the first paragraph in the cell is set to 'Bottom' in the FrameMaker Paragraph Designer panel, or if the <Cell ... > code 'Ca' option is set to `B` In these cases the baseline of the text will again be placed one third of the character pointsize *above* the top edge of the bottom cell margin. An exception to this rule occurs if the table row has a fixed or maximum height insufficient for all text to fit within the row, in which case the excess text may be hidden behind the next lower cell in the table column.[1]

Inter-paragraph spacing

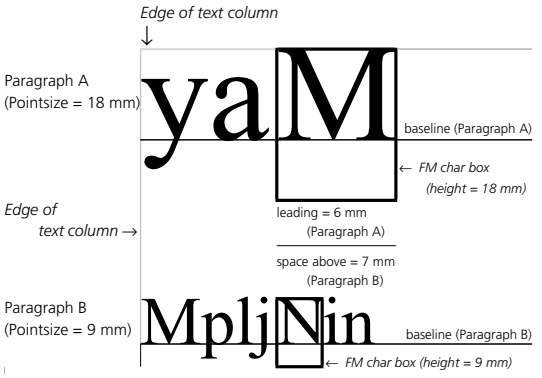


Figure 4.4 Inter-paragraph spacing

Inter-paragraph spacing within a paragraph is calculated using FrameMaker character bounding rectangles. This is illustrated by reference to Figure 4.4.

Inter-paragraph spacing is normally seen as the distance between the bottom edge of the character rectangle of the upper paragraph and the top edge of the character rectangle of the lower paragraph. This distance is determined by the sum of the leading used in the upper paragraph plus either the upper paragraph's space below value or the lower paragraph's space above value, whichever is the greater. Thus the upper paragraph's <P ... > code (or <ParaDef ... > code 'Paragraph Designer') option settings for space below (sb) and leading (l), and the lower paragraph's <P ... > code setting for space above (sa) may be used to control the inter-paragraph spacing. If feathering is switched on, then the actual inter-paragraph spacing may increase (up

1. If the fill value of the next lower cell in the table column is set to `15` (transparent), then excess text from the above cell will show through.

to the maximum value specified by the <DocDef ... /> 'maxp' option, or specified in a template file), but not decrease.

Negative values for paragraph space below and space above settings may be used to bring particular paragraphs closer together than normal, or even to make them overlap.

For many users the important inter-paragraph spacing parameter is the *baseline* offset between one paragraph and the next. This baseline offset is actually *greater* than the sum of values mentioned above, by an amount equal to one third the pointsize of the upper paragraph plus two thirds the pointsize of the lower paragraph.

Inter-line spacing

Inter-line spacing is normally determined by the pointsize of the default paragraph font plus the leading. The pointsize for the default paragraph font may be set using the Paragraph Designer, or using the <P ... > code 'p' option (see Chapter 3, page R-222 in the *Miramo Reference Guide*), or using the <FontDef ... /> 'p' option (see Chapter 4, page R-336 in the *Miramo Reference Guide*). The leading may be set similarly, using the 'l' option in place of or in addition to the 'p' option.

Figure 4.5 shows inter-line and inter-paragraph spacing when line spacing is set to be 'Fixed' (i.e. the <P ... > code 'ls' option is set to F.

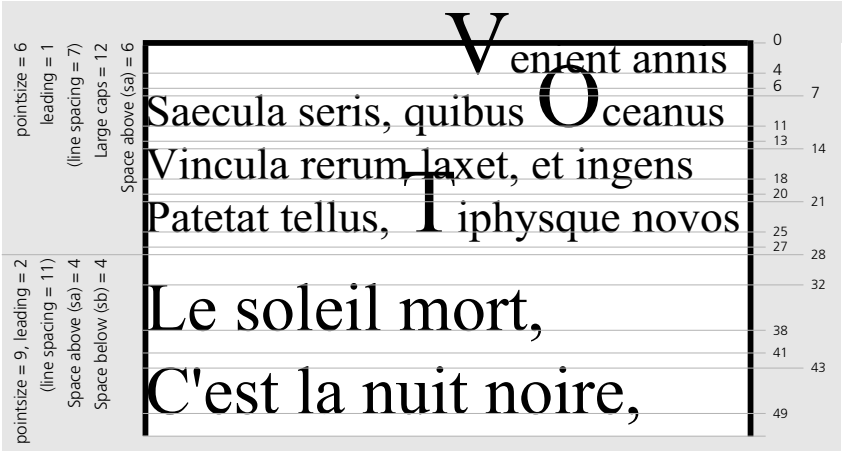


Figure 4.5 Fixed line spacing

In the case of 'Fixed' line spacing, character formats that are very much larger than the default font size may overprint text in the lines above and below. Similarly, large inline anchored frames may overprint the text above, and/or, if the anchored frame has a negative baseline offset (e.g. <AFrame bo=-12 >), the text below.

Figure 4.6 shows inter-line and inter-paragraph spacing when line spacing is set to be not 'Fixed' (<P ls=P >):

Note that in Figure 4.6 the inter-paragraph spacing has increased even though the values for leading, space below and space above remain unchanged throughout. This increase is caused by the large 'T' in the final 'Tiphysque': since the paragraph does not have fixed

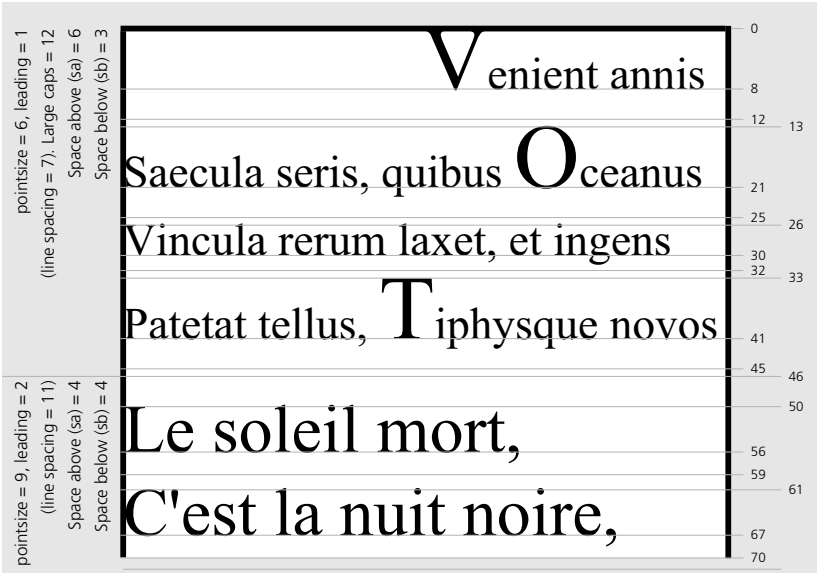


Figure 4.6 Variable line spacing

line spacing, space has been allowed below the line (for descenders) on the basis of a 12 unit pointsize.

Superscripts and subscripts

By default superscripts and subscripts are reduced in size to 80% of the paragraph font size. By default the *baselines* of superscript and subscript are raised (or lowered) by a vertical distance equivalent to 40% (or 25%) respectively of the paragraph font size. No use is made of the character bounding rectangles in calculating the vertical offsets of superscripts or subscripts.

CHAPTER 5

Books, TOCs and indexes

Documents that include tables of contents and indexes are created using book commands, either the '-Bfile' command line option or the <Doc ... > code 'Bfile' option (see the *Miramo Reference Guide*, pages R-10 and R-120), and the <Chapter ... > and <GenChapter ... > inline markup codes.

The <Chapter ... > inline markup code is described in the *Miramo Reference Guide* on pages R-89–104. The <GenChapter ... > inline markup code is described in the *Miramo Reference Guide* on pages R-161–170.

The following are distinguishing features of books:

- Books are the only way to include one or more indexes within a document.
- Books are the most convenient way to include one or more tables of contents within a document. (Tables of contents may alternatively be made using cross-references embedded within a single file.)
- Books enable breaking down very long documents into shorter component parts.
- Books enable the inclusion of different page sizes within the same multi-chapter document (each chapter may have a different page size).
- Books can be created and removed on the fly, like all other intermediate document formats.

A schematic illustrating the structure and generation process of books is shown in Figure 3.23 on page R-101 in the *Miramo Reference Guide*.

Basic books

A book is an ordered collection of documents created using the <Chapter ... > and <GenChapter ... > codes, with special capabilities for the automatic generation of tables of contents and indexes. Books enable rule based trans-chapter paragraph and page numbering.

Each chapter in a book is a separate document. All the options for each chapter document must be specified individually. A book may contain from one to several hundred chapters. A book containing just one chapter has no special benefits.

Book chapters are created using the <Chapter ... > and the <GenChapter ... > codes. The <Chapter ... > and <GenChapter ... > codes each have a mandatory 'file' option, used to specify the document name of the chapter. Example 5.1 shows the input to create a book containing four chapters named 'cover.fm', 'contents.fm', 'intro.fm' and 'cuttlefish.fm' (see lines 3, 9, 17 and 23).

A book must itself have a name. The book name may be specified on the command line as follows:

```
miramo  -Bfile book1.book   -Opdf book1.pdf     book1.mm
```

In the above the name of the book file is 'book1.book'. 'book1.mm' is the input file, e.g. as shown in Example 5.1 and 'book1.pdf' is the automatically produced output.

A book name may also be specified using the <Doc ... > code. For example, if the following were to be placed at the beginning of Example 5.1:

```
<Doc Bfile=book1.book Opdf=book1.pdf >
```

(i.e. line 2 were to be uncommented) then running the command:

```
miramo  book1.mm
```

would produce the same output. By this means, and using successive <Doc ... > codes, multiple book files can be created within a single input stream.

Example 5.1 shows input to create a minimalist four-chapter book, including a table of contents ('contents.fm', see line 9).

```
1  <!-- book1.mm         -->
2  <!--   <Doc Bfile=book1.book Pfile=book1.ps >    -->
3  <Chapter file=cover.fm >
4            <DocDef W=26mm H=40.0mm cn=1 pm=1.0mm pd=Y />
5            <VarDef fmt="V_Title">A sunny day in the sea</VarDef>
6            <P fmt=Body >COVER<P fi=2mm li=2mm p=8pt sa=10mm>
7  Sample cover page</P>
8            <VarDef fmt="V_Author">William Whale</VarDef>
9  <GenChapter file=contents.fm type=TOC StartSide=Right>
10           <GenInclude fmt="ChapterTitle" />
11           <GenInclude fmt="Heading1" />
12           <GenInclude fmt="Heading2" />
13           <VarDef fmt="V_Title">A sunny day in the sea</VarDef>
14           <DocDef W=26mm H=40.0mm cn=1 pm=1.0mm pd=Y />
15           <P fmt=Body fw=Bold sb=11pt >CONTENTS
16           </GenChapter>   <!-- Optional termination code -->
17 <Chapter file=intro.fm StartSide=Right> <!-- Default page numbering: 'Read from file' -->
18           <DocDef W=26mm H=40.0mm cn=1 pm=1.0mm pd=Y />
19           <P fmt=ChapterTitle p=10pt >Introduction<P fmt=Heading1 p=8pt >This book
20           <P fmt=Body p=8pt >It's all about fish: salmon, shark, turbot,
21 eel, a flying fish, sole, red snappers. All are my cousins ...
22           <P fmt=Heading1 P=P>My friends
23 <Chapter file=cuttlefish.fm N=Cont StartSide=Right >
24           <DocDef W=26mm H=40.0mm cn=1 pm=1.0mm pd=Y />
25           <P fmt=ChapterTitle>Cuttlefish
26           <P fmt=Body >One still, sunny afternoon I was gliding
27 along the ancient shores of Sicily ...
28           <P fmt=Heading1 P=P >Dinner<P fmt=Heading1 P=P >Breakfast
```

Example 5.1 Basic book containing four chapters[1]

Apart from font overrides, nearly every aspect of the input in Example 5.1 depends on defaults. The exception to this is that each chapter in the book has a <DocDef ... /> code setting the page width and height to the smallest permitted size, 26 mm × 51 mm, and the main flow margin, using the 'pm' option, to 1 mm (see lines 4, 14, 18 and 24 in Example 5.1). If these <DocDef ... /> codes were omitted the page size for each document would be

1. In Example 5.1, as in many other examples in the Miramo documentation, <Chapter ... > codes, and other codes, are included without corresponding termination codes. Note that if Miramo input is being generated via *mmxslt* then such termination codes *must* be included, e.g. a </Chapter> code must be included at the beginning of, or immediately preceding, lines 9 and 23 and included at the end of the file, before a </Doc> code.

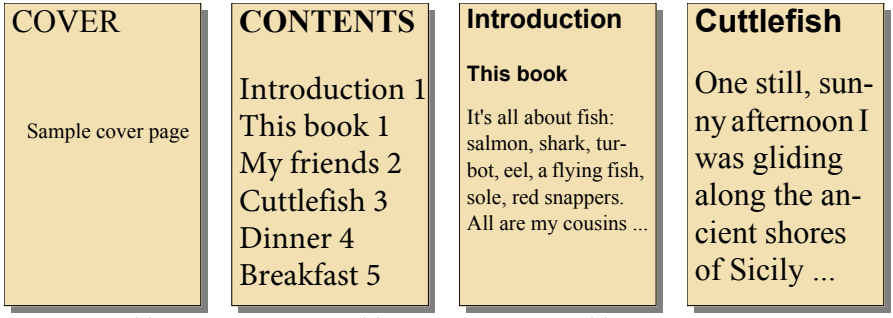


Fig. 5.1 (1) **cover.fm** (1 of 1) 1st page in book (page '1')

Fig. 5.1 (2) **contents.fm** (1 of 1) 3rd page in book (page '1')

Fig. 5.1 (3) **intro.fm** (1 of 2) 5th page in book (page '1')

Fig. 5.1 (4) **cuttlefish.fm** (1 of 3) 7th page in book (page '3')

Figure 5.1 Basic book containing four chapters (partial output from Example 5.1)

8.5 inches wide and 11 inches high, with a 1 inch margin around the main text flow. (The <DocDef ... /> code is described on pages R-319–331 in the *Miramo Reference Guide*.)

The output from Example 5.1 is shown in Figure 5.1 (first page of each chapter only). The individual chapter documents, 'cover.fm', 'contents.fm', 'intro.fm' and 'cuttlefish.fm', contain one, one, two and three pages respectively, seven pages in total. The second page of the 'intro.fm' chapter contains only the heading text 'My friends'. This is on the second page because the <P ... > code on line 22 in Example 5.1 has an override:

```
22    <P fmt=Heading1 P=P>My friends
```

i.e. the paragraph 'P', placement option is set to `P`, to force the paragraph to start on the next page.

Similarly the chapter 'cuttlefish.fm' contains three pages because line 28 contains two <P ... > codes with the paragraph 'P', placement option is set to `P`:

```
28    <P fmt=Heading1 P=P >Dinner<P fmt=Heading1 P=P >Breakfast
```

Books are always updated automatically.[1] When the 'book1.book' in Example 5.1 is updated the page count increases to a total of nine pages. This is because the 'contents.fm' and 'intro.fm' chapters are set to start on a Right page using the <Chapter ... > and <GenChapter ... > 'StartSide' option:

```
9     <GenChapter file=contents.fm type=TOC StartSide=Right>
```

and

```
17    <Chapter file=intro.fm StartSide=Right> <!-- Default page numbering: 'Read from file' -->
```

Variables in books

The <VarDef ... > codes shown on lines 5, 8 and 13 in Example 5.1, on page U-66, are included to illustrate how variables are defined in books.[2] (The <VarDef ... > and <Var ...

1. The exception to this is if the either the '-Bupdate' command line option (see the *Miramo Reference Guide*, page R-10) or an enclosing <MiramoXML ... > or <Doc ... > code has a 'Bupdate' option setting of `N` (see page R-456 and page R-121 in the *Miramo Reference Guide*).

/> codes are described on pages R-294–295 and R-293 in the *Miramo Reference Guide*.)

Variable definitions may be included at the top level anywhere after the <Chapter ... > code or, in the case of generated chapters, anywhere after the <GenInclude ... /> codes that follow the <GenChapter ... > code. If a variable is to be defined with the same value in every chapter in a book, the variable definition must be included within every chapter: there is no way to define a variable value just once on a book-wide basis.

The remainder of this section discusses some small changes that could be made to Example 5.1 to make the 'cover.fm' output look more presentable.

```
1  <Chapter file=cover.fm Cfile=${TPL}/cover_tpl.mif >
2          <VarDef fmt="V_Title">A sunny day in the sea</VarDef>
3          <VarDef fmt="V_Author">William Whale</VarDef>
4  </Chapter>
```

Example 5.2 Modified input for 'cover.fm'

Figure 5.2 'cover_tpl.mif'

In Example 5.2 the input to produce the book cover, 'cover.fm', shown in lines 3 through 8 in Example 5.1, has been changed to reference a template file, 'cover_tpl.mif', using the <Chapter ... > code 'Cfile' option (see page R-90 in the *Miramo Reference Guide*). The template document 'cover_tpl.mif', shown in Figure 5.2, comprises a single body page containing two variables, 'V_Title' and 'V_Author', and a background image. The output from Example 5.2 is shown in Figure 5.3.

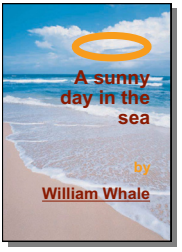


Figure 5.3 Modified 'cover.fm'

The cover template document can be extended to include any amount of extra information, e.g. date, time and copyright ownership, either on the front or on a second, verso, page. The <Chapter ... > code 'Cfile' option in line 1 in Example 5.2 causes the whole of the referenced template document to be copied into the document named by the <Chapter ... > 'file' option (see line 1 in Example 5.2). The file referenced by the 'Cfile' option must be a FrameMaker file saved in MIF format. Format definitions, e.g. <VarDef ... > codes as shown in Example 5.2, lines 5 and 8, that are included immediately after the <Chapter ... > code will override the specifications in the template document.

The equivalent procedures for using the FrameMaker GUI to create template files to control the formatting of tables of contents, body chapters and indexes is described in the following sections.

Note that the same effects may be achieved using Miramo Format Definition codes described on pages R-299 to R-413 in the *Miramo Reference Guide*.

Tables of contents

In its most basic form a table of contents simply includes a sub-set of all the paragraphs in

2. In Example 5.1 the <VarDef ... > codes are actually redundant, since there are no corresponding <Var ... /> codes located in the 'cover.fm' document, and the <Chapter ... > code does not reference a FrameMaker template document containing instances of referenced variables.

a book. In this context a paragraph comprises four separable components:

1 Paragraph body text
Referenced using the <$paratext>[1] building block.

2 Paragraph autonumber number and autonumber text
Referenced using the <$paranum> building block.

3 Paragraph autonumber number
Referenced using the <$paranumonly> building block.

4 Paragraph name
Referenced using the <$paratag> building block.

Each of the above paragraph components may be included in a table of contents by including one or more of the above building blocks in the TOC flow in the Reference pages of the table of contents file or in the template file referenced by the <GenChapter ... > 'Tfile' option. These building blocks must be included in paragraphs named *paraname*TOC, where *paraname* is the name of the paragraphs in the book components which are to be included in the table of contents. This can be illustrated with reference to Example 5.3, which specifies that components from the two paragraphs 'ChapterTitle' and 'Heading1' are to be included in the table of contents 'contents.fm', and with reference to Figure 5.6 which shows the <$paratext> and <$pagenum> building blocks in the ChapterTitleTOC and Heading1TOC paragraphs in the TOC flow in the 'content.fm' file (items **C** and **D** in Figure 5.6 on page U-71).

Figure 5.4 Modified 'contents.fm'

Almost the entire text of a book may be included in a table of contents. But usually a table of contents will contain just the paragraphs within a particular class, for example, the paragraphs that contain chapter title, heading and sub-heading text. Other tables of contents may contain the text of special figure captions or table title paragraphs. Nearly all traditional, printed tables of contents show the page number on which the listed paragraph appears in the book. Often a volume or section number and a chapter number are shown as well.

A book that includes the paragraph formats named 'Body', 'CellBody', 'Heading1', 'Heading2' and 'Title' will most likely include the text of all the 'Heading1', 'Heading2' and 'Title' paragraph formats in the table of contents, but not the text of the 'Body' and 'CellBody' paragraphs.

1. The FrameMaker GUI building blocks <$paratext>, <$paranum>, <$paranumonly> and <$paratag> correspond to the Miramo markup codes <paratext/>, <paranum/>, <paranumonly/> and <paratag/> respectively.

Creating a table of contents

There are two basic steps to creating a table of contents:

1 Use the <GenChapter ... > code (see pages R-161–170 in the *Miramo Reference Guide*), set its 'type' option to `TOC` and set its 'file' option to the name of the generated table of contents. The table of contents file name can be any name allowed by the operating system, except that it must not include single or double quotes and must not end in '.mif'.

2 Include a series of one or more <GenInclude ... /> codes (see pages R-171–172 in the *Miramo Reference Guide*) after the <GenChapter ... > code, setting the 'fmt' option on <GenInclude ... /> code to the name of a paragraph format of the text to be included in the contents.

These steps are illustrated in Example 5.1, lines 9 through 16 on page U-66, and in Example 5.3.

Included paragraph components can comprise the entire paragraph including autonumber and autonumber text, paragraph body text only or autonumber only.

Example 5.3 shows how to modify the input in Example 5.1 (lines 9 through 16 on page U-66) so that the formatting of the dynamically created table of contents is determined by a template file.

```
1  <GenChapter file=contents.fm Tfile=${TPL}/contents_tpl.mif type=TOC >
2          <GenInclude fmt="ChapterTitle" />
3          <GenInclude fmt="Heading1" />
4          <VarDef fmt="V_ChapterTitle">Contenu</VarDef>
```

Example 5.3 Modified input for 'contents.fm'

As in Example 5.2, the 'Tfile' option is used to specify the template file to copy, i.e. 'contents_tpl.mif'. The 'Tfile' option works similarly on the <GenChapter ... > code and on the <Chapter ... > code, except that the text in the main flow of an auto-generated document is always replaced whenever the book is created or re-created. In this case the template file, 'contents_tpl.mif', comprises several page layouts, including Right, Left and First master pages. The first master page is applied to the first body page in the template file. The First master page contains a background text frame containing the variable 'V_ChapterTitle'.[1]

Note that the template file used to control the formatting a table of contents *must* contain a text flow tagged 'TOC' on one of its Reference pages.

The output from Example 5.3 is shown in Figure 5.4.

The typography and layout of the entries in the table of contents is defined by formatting of the *paraname*TOC paragraphs in the TOC flow in the Reference pages of the table of contents file. This is illustrated in Figure 5.6.

1. The <P ... > codes on lines 22 and 28 in Example 5.1 (see page U-66) have been altered to have added 'P' options set to `P` to increase the total page count in the book.

In this example, the table of contents includes the text of the paragraphs referenced by the <GenInclude ... /> 'fmt' option, along with the page number, one entry for every occurrence of each such paragraph found throughout the book, in order of appearance. The layout of the table of contents entries is determined by the corresponding paragraph names which have TOC appended at the end. In this case these paragraphs are: ChapterTitleTOC and Heading1TOC.

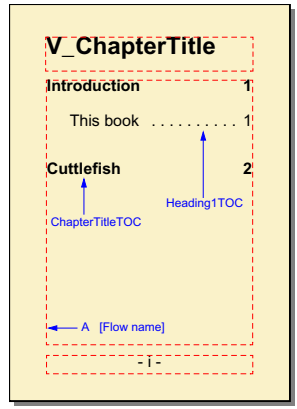


Figure 5.5 1st body page in 'contents.fm'

Three page reference building blocks may be used in a table of contents:

1 <$volnum>
Include the volume number in which this instance of the paragraph occurs.

2 <$chapnum>
Include the chapter number in which this instance of the paragraph occurs.

3 <$pagenum>
Include the page number in which this instance of the paragraph occurs.

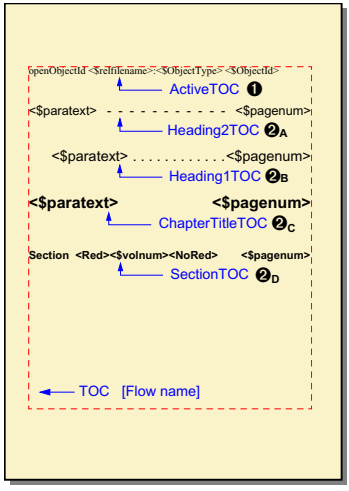


Figure 5.6 TOC Reference page in 'contents.fm' after importing formats from 'contents_tpl.mif'

A book may contain multiple tables of contents, which may be placed anywhere.

For example, some types of large catalogs are divided into multiple sections. Each section may contain multiple chapters. To include a section-specific table of contents at the beginning of each section, create special heading paragraph formats which are used only in the specific section. These paragraph formats may have identical properties to the corresponding heading paragraph formats in other sections, except that they have different names. This is easy to do using the <ParaDef ... > format definition code (see pages R-359–367), or using the Paragraph Designer in a template file.

Indexes

An index marker, its type, identifier and value are included within the input using the <IX ... > code within a paragraph or a table cell.

Indexes are sorted lists of unique marker values. If a book contains more than one marker with an identical value, the marker value is listed only once. A page reference for each marker instance is included in the index, following the marker text (marker value). The

page references exclude duplicates, except if two or more identical marker values occurring on the same page have different identifiers, i.e. the contents of the <IX ... > code are the same but the 'fmt' option is set to a different value.

Any type of marker, whether created using the <Marker ... > code or the <IX ... > code (see pages R-196–198 in the *Miramo Reference Guide*), may be included in an index. Most index entries are included using the <IX ... > code as this code supports many special index features, such as page range and sort order, not supported by the <Marker ... > code. Index levels may be separated using the <IXsub ... /> code (see page R-199).

Books may contain any number of indexes. An index entry has a default identifier of 'Index', i.e. the default value for the 'fmt' option on the <IX ... > code is Index. But values of the default 'Index' identifier are only included in an index if it is referenced by the <GenInclude ... /> code 'fmt' option. Any index identifier may be used. For example, an additional index of part numbers could be created by setting the <IX ... > code 'fmt' option to, say, PartNumber.

Creating an index

The procedure for creating an index is very similar to the procedure for creating a table of contents (see page U-70). First use the <GenChapter ... > code (see the *Miramo Reference Guide*, pages R-161–170), set the 'type' option to `IX` and set the 'file' option to the name of the index file. The type of index entries, defined by the <IX ... > code 'fmt' option, to be included in a specific index must each be specified using <GenInclude ... /> codes immediately after the <GenChapter ... > code. If no <GenInclude ... /> codes follow the <GenChapter ... > code, the generated index will always be empty. I.e. the default 'Index' index identifier will not be included in the index unless specifically requested using the <GenInclude ... /> code.

Example 5.4, a variation of Example 5.1 on page U-66, shows how to add an index, 'gen_index.fm', to a book containing the two body chapters 'intro.fm' and 'cuttlefish.fm'. The items to be included in the index are identified by <IX ... > and </IX> delimiters and the <IXsub ... /> separator in these chapters.

```
1  <Chapter file=intro.fm >
2          <P fmt=ChapterTitle>Introduction<P fmt=Heading1>This book
3          <P fmt=Body >It's all about fish: <IX>salmon</IX>, <IX>shark</IX>,
4  <IX>turbot</IX>, <IX>eel</IX>, a <IX>flying fish</IX>, <IX>sole</IX>, red snappers.
5  All are my cousins ...
6          <P fmt=Heading1 P=P>My friends<Marker fmt=Index>friends</Marker>
7          <P fmt=Body >
8  My deepest gratitude goes to <IX>cuttlefish</IX>
9  <IX fmt=Friends ix-show=N >cuttlefish</IX>, who
10 </Chapter>
11 <Chapter file=cuttlefish.fm N=Cont >
12         <P fmt=ChapterTitle>Cuttlefish
13 <IX ix-show=N ix-page=Start >cuttlefish</IX>
14         <P fmt=Body >One still, sunny afternoon I was gliding
15 along the ancient shores of Sicily ...
16         <P fmt=Heading1 P=P >Dinner
17 <IX ix-show=N ix-page=Start >meals</IX>
18 <IX ix-show=N >meals<IXsub/>dinner</IX>
19 <P fmt=Heading1 P=P >Breakfast
20 <IX ix-show=N >meals<IXsub/>breakfast</IX>
21 <IX ix-show=N ix-page=End >cuttlefish</IX>
22 <IX ix-show=N ix-page=End >meals</IX>
23 </Chapter>
```

```
24  <GenChapter file=gen_index.fm  Tfile=${TPL}/index_tpl.mif type=IX N=Cont StartSide=Right >
25          <GenInclude fmt="Index" />
26          <GenInclude fmt=Friends />
27          <VarDef fmt="V_ChapterTitle">General Index</VarDef>
28  </GenChapter>
```

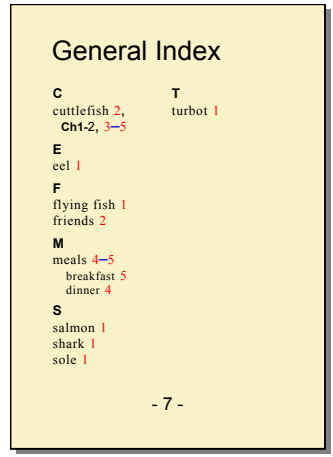
Example 5.4 Creating an index

In Example 5.4 the index chapter 'gen_index.fm' is created using the <GenChapter ... > code shown on line 24. The typography and layout rules of the index are specified in the 'index_tpl.mif' template file. The following <GenInclude ... /> codes specify that 'gen_index.fm' should include index marker types 'Index' and 'Friends'. The <VarDef ... > code at the end sets the value of the 'V_ChapterTitle' variable to 'General Index'.

Most of the index identifiers in the standard book files in Example 5.4 are type 'Index', since most are denoted by <IX and </IX> and the default type is 'Index'. Most of the index entries will appear as plain text in the body of the standard chapters, since the <IX ... > 'ix-show' option is not set to N in most cases.

Note that in Example 5.4 the terminating </Chapter> and </GenChapter> codes, marking the end of the scope of the <Chapter ... > and <GenChapter ... > codes, may be omitted.

Exceptions to the foregoing are entries that start and finish index ranges, that are second level index entries or that have a non-default type. Examples of range entries are shown on lines 13, 17, 21 and 22 in Example 5.4. Examples of multi-level entries are shown on lines 18 and 20. The only non-default index entry type is shown on line 9. The output from Example 5.4 is shown in Figure 5.7.


General Index

C
cuttlefish 2,
Ch1-2, 3–5
E
eel 1
F
flying fish 1
friends 2
M
meals 4–5
breakfast 5
dinner 4
S
salmon 1
shark 1
sole 1
T
turbot 1
- 7 -

Figure 5.7 'gen_index.fm' created using Example 5.4

The typography and layout rules for indexes are controlled by the IX flow in the index's Reference pages. An overview of how to set up index Reference pages is contained in the *FrameMaker 6.0 User Guide* on pages 345–357.

In Example 5.4 the <GenChapter ... > code (line 24) uses the 'Tfile' option to specify a file, 'index_tpl.mif', to use as a template for the typography and layout of the 'gen_index.fm' index.

Figure 5.8 shows the formatting of the special IX text flow within the reference pages of the 'index_tpl.mif' template file used to control the index layout.

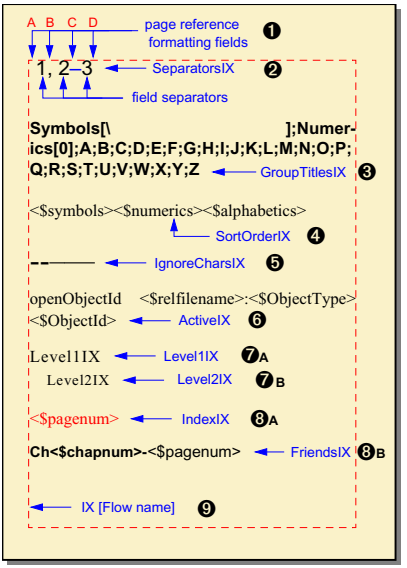


Figure 5.8 Reference page in 'index_tpl.mif'

The three digits, 1, 2 and 3 in the SeparatorsIX paragraph, shown in the top left corner of Figure 5.8 (see item in Figure 5.8), are literals used as field separators between the four fields labeled 'A' through 'D' (see item in the top left corner of Figure 5.8). The 'A' field contains the text to be placed at the beginning of the entire set of page references for an index entry. The 'B' field contains the text to be placed between each page reference or each page range reference. The 'C' field contains the text to be placed between the start and end of each page range reference. The 'D' field contains the text to be placed at the end of the entire set of page references for an index entry and at the end of each group title. Any characters, including spaces, thin spaces, non-breaking spaces, tabs, various types of dashes, dingbats, etc, may be placed within the fields 'A' through 'D'. All character attributes, such as font, color, spread, underlining, vertical shift, etc, are effective character by character within each field.

The default values for fields 'A' through 'D' are space, comma space, endash and empty. If the IX flow in the Reference pages of an index contains no SeparatorsIX paragraph, it will be created automatically with the foregoing default field values.

To create an index where the set of page references are right aligned, as illustrated in Figure 5.9, replace the default space in field 'A' with a right aligned tab.

Cuttlefish 33, 36, 291-308

Figure 5.9 Index with right-aligned page references

Assuming the index entry shown in Figure 5.9 is a level 1 index entry (i.e. a top level index entry) the markup shown in Example 5.5 could be used to produce the right-aligned effect shown in Figure 5.9.

```
1  <ParaDef fmt=Level1IX
2           an=N     sa=0pt   sb=0pt   p=9pt    l=2pt     fw=Regular
3           A=L      li=2mm   wp=N     wn=N     ff=TimesNewRoman
4  ><TabStops><TabStop A=R pos=59mm/></TabStops></ParaDef>
5  <P fmt=Level1IX >Level1IX
6  <P fmt=SeparatorsIX ><TabStops><TabStop A=R pos=59mm/></TabStops>
7  <tab/><FontRef fmt="F_Blue">*</FontRef> 1, 2&endash;3
```

Example 5.5 Creating right-aligned page references

An extra formatting feature is introduced in line 7 in Example 5.9 in the form of a font change. The effect is that the set of page references for every index entry is preceded by a string comprising a tab, a blue asterisk and a space.

The IndexIX paragraph, item **A** in Figure 5.8, specifies the formatting of page numbers referencing the default 'Index' index marker identifier. Any set of characters placed before or after the <$pagenum> building block will appear before or after the page number in

the index. Special formatting is achieved by applying one or more pre-defined character formats (or <FontRef ... > codes) to the <$pagenum> building block or to characters before or after it. Two additional building blocks, <$chapnum> and <$volnum>, may be used to include references to chapter number and volume number.

The FriendsIX paragraph, item **B** in Figure 5.8, specifies the formatting of page numbers referencing the 'Friends' index marker identifiers. This includes the fixed text 'Ch', the <$chapnum> building block, followed by a minus character, '-', all in bold, ahead of the <$pagenum> building block. In this way the page reference in the generated index for all 'Friends' entries have a distinct appearance in the generated index (see the second page reference under 'cuttlefish' in Figure 5.7).

The index Reference page may contain several paragraphs of type *string*IX, where *string* is the name of each index identifier type included in the index, i.e. the value of the <IX ... > code 'fmt' option.

Multiple indexes

Indexes may be included anywhere within a book. A book may contain any number of indexes. For example a special 'Index of Friends' may be created by adding the input shown in Example 5.6 to the end of the input shown in Example 5.4 (see page U-73).

```
1   <GenChapter file=friends_index.fm  Tfile=${TPL}/index_tpl.mif type=IX
          N=Cont StartSide=Next >
2       <GenInclude fmt=Friends />
3       <VarDef fmt="V_ChapterTitle">Index of Friends</VarDef>
```

Example 5.6 Adding another index

The output from Example 5.6 is shown in Figure 5.10.

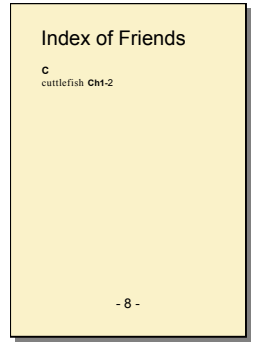


Figure 5.10 'friends_index.fm' created using Example 5.6

CHAPTER 6

Cross-references and hypertext

This chapter describes how to use markup codes to set up cross-referencing and/or hypertext linking in and between documents.

It also describes how to create documents for distribution in PDF or view-only format, where cross-references and hypertext markup codes are to be mapped to hyperlinks and PDF notes in the PDF output document, or hyperlinks and alert boxes in a view-only Frame document.

Cross-references

Cross-references consist of a cross reference *destination* (created using the <MkDest ... /> code, see page R-216 in the *Miramo Reference Guide*) and one or more cross-reference *instances* (inserted using the <XRef ... /> code, see page R-296) which refer to that destination.

The appearance and wording of the cross-reference instance is determined by a cross-reference *format*, defined using the <XRefFmtDef ... > code (see page R-410). The <XRefFmtDef ... > definition includes text and special 'building block' codes which indicate how the cross-reference is to be constructed - for example, using the paragraph text or page number containing the destination identifier.

A cross-reference instance is linked to a particular destination by a unique *destination identifier*, supplied using the <MkDest ... /> and <XRef ... /> 'id' option.

The sections below describe using Miramo markup codes to establish a cross-reference *destination* (Example 6.1), a cross-reference *instance* (Example 6.2), and a cross reference *format definition* (Example 6.3).

Creating a cross-reference or hyperlink destination

A cross-reference *destination* is the location of the point referred to by the cross-reference, and is created using the <MkDest ... /> code. The 'id' option specifies a destination identifier, and must be unique within a document.

By default, if no 'fmt' option is specified, the <MkDest ... /> code will create a destination identifier suitable only for reference by the <XRef ... /> inline markup code. Setting the 'fmt' option to `H` will create a destination identifier suitable only for hyperlink commands created using the <HyperCmd ... > code 'jumptodest' option. Setting the 'fmt' option to `B` (both cross-reference and hypertext) will create a destination identifier which may be referred to either by the <XRef ... /> code, or by the <HyperCmd ... > code.

The example below uses 'HM001' as the unique destination identifier to be referred to later either from a cross-reference instance, such the one shown in Example 6.2, or from a hypertext link command such as the one shown in Example 6.4.

```
<P fmt=head2>
<MkDest id="HM001" fmt=B />House Music
<P fmt=output>
Combines funk with hi-tech pop.
```

Example 6.1 Using <MkDest ... /> to create a hidden cross-reference destination identifier

The output from Example 6.1 is shown in Figure 6.1. Note that the <MkDest ... /> code creates a hidden marker in the output document.

House Music

Combines funk with hi-tech pop.

Figure 6.1 Output from Example 6.1

Creating a cross-reference instance

Cross-reference instances are inserted using the <XRef ... /> code. The destination and instance are linked together using the destination identifier specified using the 'id' option on both the <XRef ... /> code and the <MkDest ... /> code to which it refers.

In the output document, the cross-reference instance created using the <XRef ... /> code is replaced by a reference to the destination identifier.

If the 'file' option is omitted from the <XRef ... /> code, the current document is assumed to contain the destination identifier. Otherwise the 'file' option gives the pathname of the document containing the destination identifier.

Example 6.2 shows a cross-reference instance which refers to the destination identifier given in Example 6.1, using the 'id' option set to HM001:

```
<P fmt=output>
Funk is dance music of US origin.
<XRef fmt="See Page" id="HM001" file="../music.fm" />
```

Example 6.2 Using <XRef ... /> to insert a cross-reference

Assuming that the cross-reference format is defined as shown in Example 6.3, the output from Example 6.2 could be as shown in Figure 6.2 at right:

Funk is dance music of US origin. See also House Music on page 78.

Figure 6.2 Output from Example 6.2

Cross-reference formats

The appearance and wording of a cross-reference instance in the output document is determined by the cross-reference format, specified using the <XRef ... /> 'fmt' option.

Cross-reference formats use text and special 'building block' codes which indicate which information is to be extracted from the destination point in the document.[1] For example, a commonly used building block code is <paratext/>, which is replaced by the text of the paragraph containing the destination identifier.

The cross-reference formats may either be defined within the Miramo document using the <XRefFmtDef ... > code, or may be imported from a FrameMaker template file[2].

Here's the cross-reference format definition for the 'See Page' format used in Example 6.2

1. A complete list of cross-reference format building blocks is given on page .

to produce the output in Figure 6.2:

```
<XRefFmtDef fmt="See Page">
See also <FontRef fmt=F_LinkText><paratext/></FontRef> on page <pagenum/>.
</XRefFmtDef>
```

Example 6.3 Using <XRefFmtDef ... > to define the 'See Page' cross-reference format

To illustrate the use of cross-reference format building blocks the following table gives some examples of cross-reference format definitions, together with the kind of cross-references that they might produce. Some of the cross-reference format definitions include the <FontRef ... > markup code to emphasize words within the replacement text:

Format	Result
page <pagenum/>	page 23
Chapter <chapnum/>, "<paratext/>"	Chapter 2, "The Arts"
See '<paratext/>' in chapter <chapnum/>, <FontRef fmt=italic><paratext fmt="ChapTitle"/>	See 'Mona Lisa' in chapter 2, *The Arts*
See <FontRef fmt=bold>page <pagenum/></FontRef>	See **page 23**

Cross-references in view-only or PDF documents

Cross-references established as indicated above are automatically mapped to hyperlinks when the output document is a view-only or PDF document. In this case, the text of the cross-reference is 'active' so that clicking on the cross-reference text causes a jump to the specified destination identifier.

Hypertext commands in view-only or PDF documents

The <HyperCmd ... > and <MkAlert ... > markup codes may be used to insert hidden hypertext commands which are active in a view-only or PDF documents (see Creating view-only or PDF documents on page U-80).

The <HyperCmd ... > code (see page R-173) may be used to jump to a different location, to launch an external application or to open a URL. The <MkAlert ... > code (see page R-214) will display an alert message in a view-only Frame output document, and will be mapped to a PDF note in a PDF output document.

The hypertext command is executed when the user clicks on the text surrounding the hypertext markup code (the 'active area'). An active area may be the word, phrase, or entire paragraph containing the hypertext markup code: the <Font ... > .. </Font> or <FontRef ... > .. </FontRef> code pairs can be used to delimit the extent of the active text by making the text around the hypertext markup code appear visually different from the surrounding text.

Hypertext link commands may create a link to a destination identifier specified using the <MkDest ... /> code (see *Creating a cross-reference or hyperlink destination* on page U-77).

Example 6.4 illustrates using the <HyperCmd ... > 'jumptodest' option to link to the des-

2. See the '-Tfile' and '-Cfile' command line options on page R-19, and the <Chapter ... > and <Doc ... > code 'Tfile' options on pages R-89 and R-113 for information on importing format definitions from Frame template files.

tination identifier created in Example 6.1:

```
<P fmt=output>
Funk is dance music of US origin.
See also
<Font fmt=F_LinkText><HyperCmd jumptodest="HM001"
        file="../music.fm"/>House Music</Font>.
```

Example 6.4 Using <XRefFmtDef ... > to define the 'See Page' cross-reference format

The output from Example 6.4 is shown in Figure 6.3 at right.

> Funk is dance music of US origin. See also House Music.

Figure 6.3 Output from Example 6.4

Comparing this with the cross-reference created in Example 6.2, note that it is not possible to include page numbers or paragraph text using the <HyperCmd ... > 'jumptodest' option, and that the active area for the hyperlink is restricted to the red text, rather than the complete cross-reference text shown in Figure 6.2.

It is not always necessary to use the <MkDest ... /> code in conjunction with the <HyperCmd ... > code when creating link commands, as it is possible to use the <HyperCmd ... > 'jumptopage' option to create a link to the a specified page (firstpage, lastpage, nextpage, prevpage or a page number), or to use the 'jumpback' option to jump to the previously viewed location.

The <HyperCmd ... > 'openURL' option allows a URL to be opened, and the 'system' option will launch an external application. These and other options are documented on page R-173.

The <MkAlert ... > option may be used to create an alert window in view-only documents, or a PDF note in PDF output documents, when a user clicks on the active area.

Creating view-only or PDF documents

Cross-references and hypertext navigation commands, created as described above, are automatically mapped to hyperlinks in view-only or PDF output documents.

The Miramo command line options '-Oviewfm' and '-Oviewmif' (see page R-11), and the related <Doc ... > code 'Oviewfm' and 'Oviewmif' options (see page R-113) may be used to create view-only output documents suitable for viewing with FrameMaker or FrameViewer.

The Miramo command line options '-Opdf' and '-PAfile' (see pages R-12 and R-15) may be used to create PDF output files or distiller-ready PostScript files which preserve cross-references and hypertext commands.

CHAPTER 7

Controlling page layouts

Introduction

This chapter describes how to control the page layout formats produced by Miramo output using the 'master' and 'body' pages contained in a Frame template file, or master pages defined using the <PageDef ... > format definition code (see pages R-352–352).[1] Master and body pages defined in a template file may be incorporated in a Miramo document using the Miramo '-Tformats' command line option, or the <Template ... /> code 'l' or 'L' (page layout) options.

There are two different approaches to controlling page layout usage:

- Use default rules for applying the body page and 'Right' master page (or in the case of 'double-sided' documents, 'Right' and 'Left' master page) layouts from a template file.

  In this case, the body pages in the template file determine the page layouts for the initial pages produced by Miramo. Thus, if the template file contains three body pages, the first three body pages of the generated Miramo document will adopt the page layouts of the three template file body pages. The default page layout for follow-on pages is determined by the special master page 'Right' (or in the case of 'double-sided' documents, the special master pages 'Right' and 'Left').

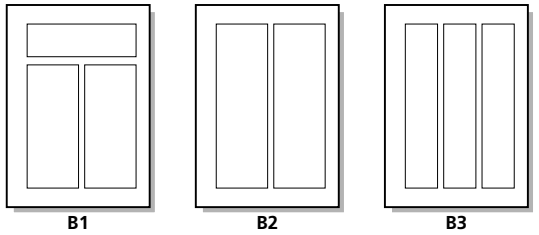
- Use the Miramo <MasterPageRule ... /> code to apply master page layouts (defined in a Miramo template file, or using the <PageDef ... > format definition code, or a combination of these) as required, on a data-driven basis.

The above approaches may be combined to achieve desired effects.

Simple template file usage: 'single-sided' documents

By default Miramo uses the layout of the body pages in a template file to determine the layout of the output document. Thus for example if a template file, 'one.mif', contains three body pages, labeled B1, B2 and B3, laid out as shown in Example 7.1.

1. Body pages contain the paragraphs of text and graphics comprising the document content. Master pages are used to design page layouts that may subsequently be applied to specified body pages. The master pages hold background running header/footer text and graphics, together with text column layouts.

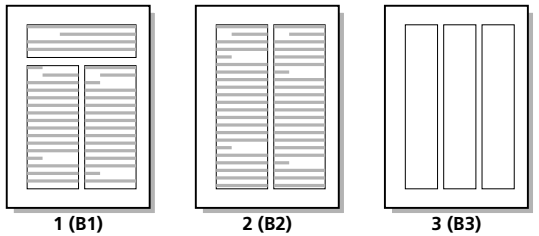


Example 7.1 Body pages in template file 'one.mif'

running the command

```
miramo -Tfile one.mif -Omif out.mif data.mm
```

will result in the first three pages of the output file inheriting the page layouts of the template file, as shown in Example 7.2 below.



Example 7.2 Body pages in output file 'one.mif'

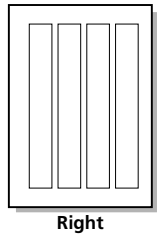
Note that if the output document is smaller than the template file, it may contain one or more trailing blank pages. Provided the document properties of the template file include the setting 'Delete Empty Pages' these empty pages will disappear after the file 'one.mif' has been opened and saved by FrameMaker. Alternatively, running Miramo with the '-Ofm' or '-Oviewfm' command line options will cause these pages to be deleted automatically if the 'Delete Empty Pages' document property is set in the template file.

Note also that though the body pages shown in Example 7.1 above contain no text, it can convenient to have sample text and tables in a template document. A template document may contain any text or graphics on its body pages. This text or graphics will always be ignored.

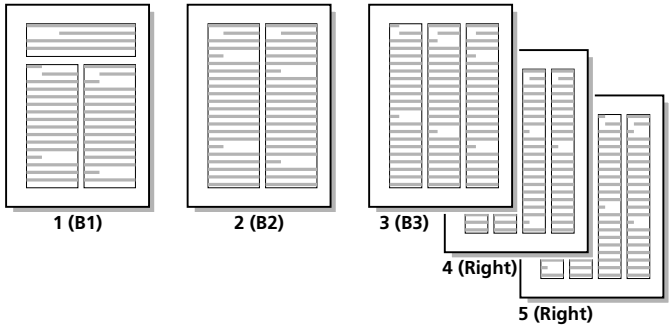
Following from the above, it can be highly convenient to 'recycle' Miramo output files into template files. This can make it easier to design paragraph and other formats interactively to the ultimate document design requirements. The template document effectively becomes a sample of the output required. Changing formats in the template gives immediate feedback regarding the appearance of production documents.

In the common case where the output document will be larger than the template document, the layout of the additional pages will by default be determined by the layout of the template document 'Right' master page, assuming a 'single-sided' template.

Suppose for example the 'Right' master page of 'one.mif' is laid out as follows:



Then the output document would look like this, assuming 'one.mif' is otherwise unchanged:



Example 7.3 Additional body pages in output file 'one.mif'

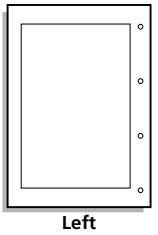
The three body pages in 'one.mif' determine the layout of the initial three pages of the output. The following two pages in the output file 'one.mif' (and any further pages) will use the four-column layout of the 'Right' master page in 'one.mif'.

Simple template usage: 'double-sided' documents

Making a document 'double-sided' changes the way additional pages are added by default. If the first additional page produces a current page count which is even, then a 'Left' master page is added, otherwise a 'Right' page is added. Thereafter data is formatted into 'Right' and 'Left' master page formats in alternating sequence.[1] 'Double-sided' documents always contain both a 'Left' and a 'Right' master page.

Suppose that we have a template file called 'two.mif' identical in every way to 'one.mif', except that it is a double-sided document and has a 'Left' master page like this:

1. This assumes that the template document is set to have its first page as a 'Right' page. To keep the following discussion as simple as possible, this assumption is made throughout the remainder of this chapter. If the first page is set to be a 'Left' page everything said about 'Left' and 'Right' page ordering is reversed.



(The 'holes' in the above, and the following, are intended to cue the orientation of the page and are not part of the page layout.)

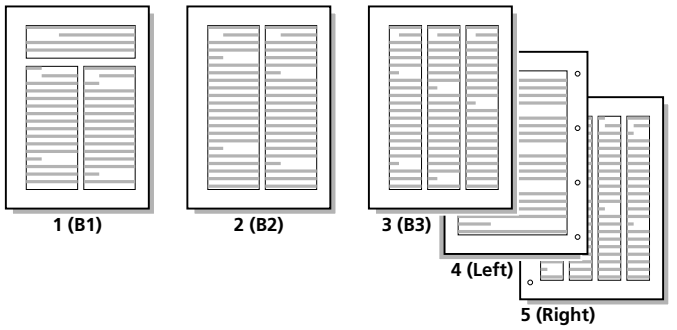
Now if we run the command:

```
miramo -Tfile two.mif -Omif out.mif data.mm
```

or:

```
miramo -Tfile two.mif -Tformats L -Omif out.mif data.mm
```

we will get the output shown in Example 7.4 below.



Example 7.4 Additional body pages in output file 'two.mif'

The first three pages of output in Example 7.4 are identical to the first three pages in Example 7.3. Changing the template document setting to 'double-sided' makes no difference to their appearance: the left and right margin offsets do not reflect the differences shown between those on left and right pages.

Simple input

The Miramo input for Example 7.4 above could have the following form:

```
<P fmt=Title>
Advances in Earthquake Warning
<P fmt=Authors>
Atlas B Mundi
<P fmt=Abstract>
A detailed review of recent developments in . . .
. . . .
. . .
<P fmt=Head1 P=C>
Introduction
```

```
<P fmt=Body>
Since the early days of yesteryore there have
been . . .
```

Example 7.5 Simple input

This input will place the article title, author and abstract in the first, spanning text column of the page labeled B1 (in Example 7.4). Note that the size of this column should be large enough to hold the largest amount of text that may be required. The input line:

```
<P fmt=Head1 P=C >
```

directs that the text following will begin at the top of the next connected column, in this case the column at the lower left side of the page labeled B1.

Creating an embedded table of contents

At this point we will divert to showing a practical example of using paragraph positioning options to achieve a special effect.

Instead of starting immediately with the body of the article, we may wish to place a table of contents in the column at the lower left side of the page labeled B1. To do this we could create a macro using *mmpp*, create a temporary file containing the markup and data for the table of contents, using the *mmpp* <@system> command, and include this file using the Miramo <Include ... /> inline markup code.

First of all we need to define a flag to signal the start of the table of contents and define a temporary file name; and then define a macro. We will assume that all paragraphs of type 'Head1' are to be included in the table of contents.

The *mmpp* coding for this is shown in Example 7.6 below.

```
<#define>          TOCfile        TOC
<@macro>          Head1                    {
                  @--- First argument is heading text
                  @--- Second argument can be <#xP> options
                  <P fmt=Head1 $2>
$1<MkDest fmt=X id=TOC.$1 />
                  @--- Write TOC entry to <#TOCfile> ready for inclusion later
                  <@write> <#TOCfile>{
                  <P fmt=TOC1>
$1
                  <XRef fmt=pagenum id=TOC.$1 />
                  }
}
```

Example 7.6 *mmpp* table of contents macro

For this macro to be effective in the way intended it must be included within the input stream, and all headings must be marked up thus:

<¦Head1> *Heading text*

For example, input could have the form shown below:

```
<@include> TOCmacros
<P fmt=Title>
Advances in Earthquake Warning
```

```
<P fmt=Authors>
Atlas B Mundi
<P fmt=Abstract>
A detailed review of recent developments in . . .
. . . .
. . .
<P fmt=TOCHead P=C>
@--- Include TOC data & remove temporary file
<Include file=<#TOCfile> />
<!--- XXXCRIN System
<?System>rm "<#TOCfile>" ?>
<¦Head1> Introduction ¦ P=C
<P fmt=Body>
Since the early days of yesteryore there have
been . . .
<¦Head1> Early history
<P fmt=Body>
Since the days of Babel there has been much talk
. . .
<¦Head1> Developments in Russia
. . .
```

Example 7.7 Miramo input for table of contents

The above input needs to be processed in two passes, *viz*.

```
mmpp  data.mmp  >  data.mm
miramo  -Tfile  two.mif  -Omif  out.mif  data.mm
```

As usual, the template file, 'two.mif' must contain format definitions for the 'pagenum' cross reference, as well as any paragraph formats required (Title, Author, Head1, TOC1 etc).

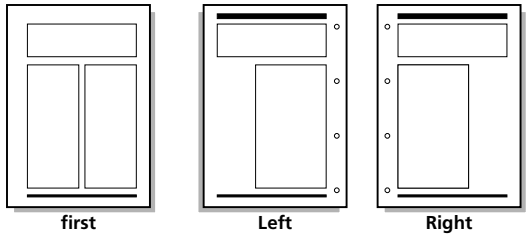
The foregoing exploits the difference between the *mmpp* <@include> code and the Miramo <Include ... /> inline markup code: each works in the same way, but since *mmpp* does not recognize <Include ... />, it is effective only during the second, Miramo pass.

Page breaks and 'filler' pages

A paragraph, a table or a row within a table may be forced to appear at the top of a page, or at the top of a 'Left' or a 'Right' page. If the current position is on the first page of a document, or on a 'Right' page, and the current paragraph is placed at the top of a 'Right' page, using the <P ... > code option settings 'fmt=*paratag* P=R', then a blank, or 'filler", 'Left' page will be output before the 'Right' page.[1] This is illustrated below.

Suppose a template file, 'three.mif', has a first body page, and a 'Left' and a 'Right' master page defined as shown:

1. This assumes double-sided document is being created. If the document is single-sided, then requests to place an object at the top of a 'Left' page are treated simply as top of page requests. That is '<P ... > fmt=*paratag* P=L' is equivalent to '<P ... > fmt=*paratag* P=P', etc.

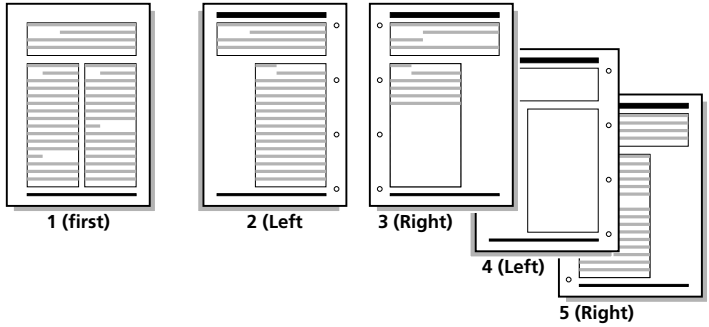


Example 7.8 'Left' and 'Right' pages in template file

and we run the command:

```
miramo -Tfile three.mif -Omif out.mif data.mm
```

Then, assuming we have a 'P=R' paragraph option half way down the third page of formatted input, we will get output as shown in Example 7.9 below.



Example 7.9 Output with a blank 'Left' page

The text at the end page 3 in Example 7.9 above is followed by a paragraph with a 'P=R' option set. This paragraph is placed at the top of page 5, and page 4 becomes a blank 'Left' page. This blank 'Left' page is the same as any preceding 'Left' page (i.e. it contains the same headers and footers), except that there is nothing in the main textflow.

Note that if the paragraph with a 'P=R' option set were on a left page, then no filler page is included, since the next page is naturally a right page.

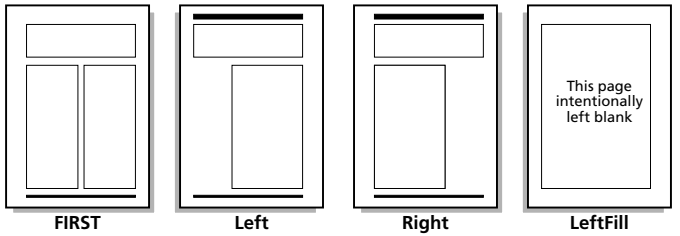
Using the <MasterPageRule ... /> code to control filler page usage

It may be required to start a new section of a 'double-sided' document on a right page and, where necessary, have a filler left page. However it may also be required that the filler left page has a different format to a regular left page. For example the filler left page may need to omit the running header and footer, and include the words 'This page intentionally left blank', or include detailed terms and conditions of supply, suitable for the

reverse side of an invoice.

To do this we need to use the <MasterPageRule ... /> code, as illustrated in the remainder of this section.

As a first step we create a special template document, with two extra master page formats, 'FIRST' and 'LeftFill', defined:[1]



Example 7.10 Template with a 'filler' master page

All the master pages in this template document must have at least one text column containing a tagged flow. Normally these columns will be autoconnected and will have 'Flow Tag' set to 'A'.[2] This rule applies in this case to the master page 'LeftFill', even though it will only be applied to pages which have no text in the autoconnected flow. If fixed text is required on master pages, it must be placed in a 'Background' text column.

Having created an appropriately formatted template document we need to ensure that the markup in the Miramo input file will produce the correct results using the <MasterPageRule ... /> code.[3] There are three rules which are important here:

1 The <MasterPageRule ... /> 'Page=*pagename*' used with any of the <P ... > 'P=P', 'P=L' or 'P=R' (start at top of page) options results in the master page *pagename* being applied to the current page (i.e. the page which the paragraph is now at the top of) regardless of any previous <MasterPageRule ... /> setting.

2 The <MasterPageRule ... /> 'Page=*pagename*'> option, *not* used with the <P ... > code 'P=P', 'P=L' or 'P=R' (start at top of page) options, applies the master page requested to the *next* page, i.e. the page following that containing the <P ... > code, unless this is overridden by Rule 1 above.

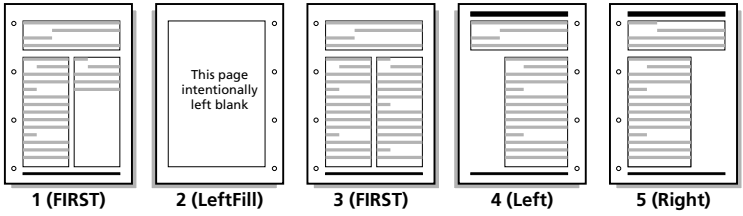
3 The rule for applying master pages to follow-on pages can be specified by using the <MasterPageRule ... /> 'oddPages', 'evenPages', or 'allPages' options, in combination with the 'Page' option.

1. In the following discussion the names of the master pages 'FIRST' and 'LeftFill' are arbitrary. *Any* master page names and formats may be used in their place (provided the names do not contain a colon or a space).

2. By default Miramo uses flow 'A'. This can be changed using the '-Tflow' command line option (see Chapter 2, 'Running Miramo', on page R-20 of the *Miramo Reference Guide*).

3. See the *Miramo Reference Guide*, Chapter 3, pages R-220–230

Based on the foregoing, we could produce the first five pages of our output document in the format shown below



using markup like this:[1]

```
<P fmt=SectionTitle P=P>
  <MasterPageRule Page=FIRST allPages=Default />
text... tables ... graphics
...
<P fmt=LeftFillPara wp=Y>                          <!-- Last para on page 1 -->
  <MasterPageRule Page="LeftFill"/>
<P fmt=SectionTitle P=R>                           <!-- First para on page 3 -->
  <MasterPageRule Page="FIRST"/>
text... tables ... graphics
```

The 'allPages=Default' option above specifies that the default page layouts for follow-on odd and even numbered pages should conform to 'Right' and 'Left' master pages respectively. Since this is the default rule, this option could be omitted.

If the last paragraph preceding page 3 occurred on page 2 instead of page 1, then page 2 would be a normal 'Left' page. Page 3 would initially be a 'LeftFill' page, but because its top of page paragraph specified using a 'FIRST' master page, the 'LeftFill' format would be overridden by the 'FIRST' format.

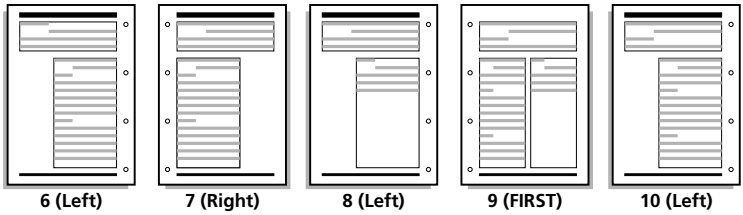
This scenario occurs between pages 8 and 9 in the next illustration, using exactly the same markup:

```
text... tables ... graphics
...
<P fmt=LeftFillPara wp=Y>
  <MasterPageRule Page="LeftFill"/>
<P fmt=SectionTitle P=R>
  <MasterPageRule Page="FIRST"/>
text... tables ... graphics
```

In this case the output will look like this:

1. Note the 'LeftFillPara' paragraph format on line 4 in the example below. This paragraph should be empty and guaranteed to remain with the preceding text or table. For example, it could be 2 points with a negative space above and below. It should only have a 'wp=Y' setting if it is preceded by text.

If the 'LeftFillPara' paragraph is preceded by a table or an anchored frame, the table or anchored frame should be anchored within its own special anchor paragraph (always good practice). A table should have a negative space below setting. This way the 'LeftFillPara' paragraph will be located behind the last row of the table, or possibly higher. In the case of an anchored frame the anchor paragraph should have a negative space below setting to achieve an equivalent effect.

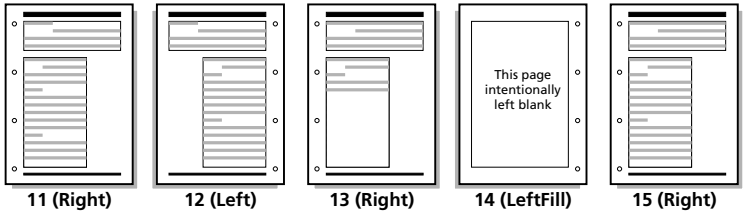


i.e. no filler page will be produced.

As a variation on the above, if a 'Right' page should follow a 'Right' page, in place of a 'FIRST' page. In this case, use

```
text... tables ... graphics
...
<P fmt=Body wp=Y>
  <MasterPageRule Page=LeftFill/>                    <!-- Last para on page 13 -->
<P fmt=SectionTitle P=R>
  <MasterPageRule Page=Right />                      <!-- First para on page 15 -->
```

to produce:



Example 7.11 'Left' and 'Right' pages in template file

Again, the 'LeftFill' page will be output only if it is preceded by a right page.

The essential trick in the foregoing is to immediately precede the request for a right page with a paragraph specifying the name of the master page for the optional left page. The intricacies of doing so may be hidden from the data extraction routines by using a *mmpp* macro.

```
<@macro> SectionTitle {
<P fmt=Body wp=Y>
  <MasterPageRule Page=LeftFill />
<P fmt=SectionTitle P=R >
  <MasterPageRule Page=FIRST oddPages=Default evenPages=Default />
$1      @-    Section title text will be substituted here
}
```

With this macro it is only required to enter the line

```
<¦SectionTitle> Section title text
```

to get filler pages working correctly before new sections. The 'oddPages' and 'evenPages'

options in the above macro are not necessary if they have been set previously. However they may be included to override preceding changes: re-iteration of this option is always harmless and often safest.

Note that the principles discussed above may be applied in ways that are considerably more complex than the foregoing illustrations show. A template document may contain several dozens of master pages, any of which may be applied at any point. Similarly the follow-on rules specified by the <MasterPageRule ... /> 'oddPages', 'evenPages' or 'allPages' options may be changed at will.

Adding disconnected pages with <DPage ... />

The <DPage ... /> inline markup code allows the addition of a disconnected page either before or after a specified paragraph (see the *Miramo Reference Guide*, Chapter 3, pages R-127–129). In general the <DPage ... /> code is much less useful and flexible than the <MasterPageRule ... /> 'Page' option.

The main issues concerning whether to use the <DPage ... /> or the <MasterPageRule ... /> 'Page' option are summarized below.

- Every occurrence of a <DPage ... /> inline code is 'effective': any number of <DPage ... /> codes may be used within a single paragraph, or within several paragraphs on a page. Each instance will result in a disconnected page being added. This is the major advantage of the <DPage ... /> code, which may mandate its use in certain applications.

  In the case of the <MasterPageRule ... /> 'Page' option, only the last occurrence on a page is 'effective'. (The exception to this rule is when the <MasterPageRule ... /> 'Page' option is used in combination with the <P ... > 'P' option being set to any one of the following 'top of page' settings: `R`, `L` or `R`. In this case the first occurrence of the <MasterPageRule ... /> 'Page' option is also effective, and the specified master page is applied to the current page, rather than the following page.)

- Miramo input will always wrap around disconnected pages added using <DPage ... />.

- Using either the <DPage ... /> inline code or the <MasterPageRule ... /> 'Page' option requires that Miramo is run using a template file (i.e. using the '-Tfile' command line option or the <Template ... /> format definition code).

<DPage ... /> is useful in cases where it is needed to place one or more full-page graphics or text boxes (for example advertisements or parts diagrams) which have multiple 'anchor' points within a single input page. The limitation here is the number of master pages which can be included in a single document.[1]

Example 7.12 illustrates using the <DPage ... /> code:

```
1  <Units fmt=in/>
2  <P fmt=Body>
3  This is text on the first page. If there is more text and graphic
4  objects than will fit on the first page, the additional material
```

1. A FrameMaker template file may contain hundreds of Master Pages.

```
will wrap around the disconnected page, on to the third page.
<PageDef fmt=Hello>              <!-- Create disconnected page layout -->
        <!-- Code within <mmDraw> and </mmDraw> puts a box with curved -->
        <!-- corners around the text column on the disconnected page. -->
        <!-- (May be omitted.) -->
        <mmDraw>
                <Rectangle pw=9pt pen=0 radius=0.6in L=1in T=2in W=6in H=8in />
        </mmDraw>
        <TextFrameDef type=B L=1 T=2 W=6 H=8 fill=15 >
        <P fmt=Body />
        <P fmt=Body A=C p=24 ff=Helvetica fw=Bold sa=1in>
Oranges are good for you!
        </TextFrameDef>
</PageDef>
<!-- Apply disconnected page after current page -->
<DPage fmt=Hello P=A >
<P fmt=Body P=P>
This text will be on a page following the disconnected page.
If this paragraph did not have a <FontRef fmt=F_NoLang >&#x2018;</FontRef>P=P&#x2019;
                option, then it would
be on the first page.
```

Example 7.12 Using the <DPage ... /> code

Putting the text contained in Example 7.12 in a file called ‘dpage’ and running the command:

```
miramo -Ofm dpage.fm dpage
```

will produce a document containing three pages. The second page will be a page containing the text ‘Oranges are good for you’ placed within a rectangle with rounded corners. If the <DPage ... /> code includes a ‘P=B’ option, or no ‘P’ option, then the first page would contain the text ‘Oranges are good for you!’.

General Index

Numerics

A

B

C

R

S

T

V