Harlequin Style and Publications Guide

Version 1.7

Eye have a spelling checker, it came with my PC;
It plainly marks four my revue mistakes I cannot sea.
I've run this poem threw it, I'm sure your please to no,
It's letter perfect in it's weigh, my checker tolled me sew!

A guide to the writing and formatting styles used by the Documentation Group, together with a discussion of many of the internal procedures of the group. This guide is of most use to members of the Documentation Group, although it may be of use to other Harlequin employees on occasion.

Contents

harlequin

Copyright and Trademarks

Harlequin Style and Publications Guide

Version 1.7

September 1997

Part number: HQN–1.7–PG

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
U.K.

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
U.S.A.

telephone +1 617 374 2400
fax +1 617 252 6505

Electronic Access:

http://www.harlequin.co.uk/
http://www.harlequin.com/

Contents

Preface

This *Harlequin Style and Publications Guide* lists solutions to documentation style issues for Harlequin documents. Most style issues are resolved by standard reference books. This *Guide* includes solutions that differ from those reference works or are not covered in those works. It also includes solutions to common issues in Harlequin documentation so that writers need not consult those longer reference works.

The standard reference works that form the basis for Harlequin style are the following:

- *The Chicago Manual of Style*, 14th edition (Chicago: The University of Chicago Press, 1993).

  On some issues this is better adapted to university press publishing than software documentation.

- *The Random House Webster's College Dictionary* (New York: Random House, 1992).

  This serves as the basis for the Harlequin use of American spellings.

- *Hart's Rules for Compositors and Readers at the University Press Oxford,* 39th edition (Oxford: Oxford University Press, 1983).

  This is more concise than Chicago and serves to explain transatlantic differences in usage.

The Harlequin documentation group uses Adobe Systems Incorporated's FrameMaker software for the preparation and electronic representation of its documents.

This *Guide* is divided into four chapters and a glossary.

Chapter 1, "Style" lists some style issues in alphabetic order.

Chapter 2, "Formats" describes the formats of the current Harlequin book design.

Chapter , "Formatting Lisp Code" describes some simple methods for formatting Lisp code in documentation.

Chapter 3, "Procedures" describes common documentation procedures, such as using HOPE.

Chapter 4, "Checklist for Finalizing Documents" describes each step in preparing a document for publication.

The Glossary lists terms alphabetically for preferred spelling, capitalization, acronym use, and other brief issues.

1 Style

This chapter contains an alphabetical listing of various style issues and their resolutions.

1.1 Abbreviations and acronyms

An abbreviation is a contraction of a word or phrase into a shortened form. Usually (but not always) an abbreviation is formed by taking the first letter of each word in the phrase. Common abbreviations formed in this way are sometimes referred to as “initialisms”. An acronym is an abbreviation formed in this way which itself forms a pronounceable word.

In general, acronyms and abbreviations formed from the first letters of a phrase should be treated in the same way. Note that there are also many circumstances where abbreviations should be avoided completely.

1.1.1 Abbreviations that should be avoided

Avoid Latin abbreviations and “informal” abbreviations that are not formed from initial letters of a phrase in most text. Table 1.1 lists some of the more common of these.

Table 1.1 Expansions for Latin abbreviations

Use this expansion …	… instead of this abbreviation
for example	e.g.
and so forth and so on	etc.
that is	i.e.
a cross-reference	q.v.
information	info
document, documentation	doc
software	s/w

1.1.2 Use of abbreviations and acronyms

In general, spell out an acronym or abbreviation in the first text reference. For example:

> The functions are stored in a DLL (dynamic link library).

Give the acronym or abbreviation itself, then spell out the name in parentheses: this is the standard Chicago advice. Alternatively, you can sometimes reverse the order, giving the expansion in the main text and the acronym or abbreviation in parentheses. If you do this, then add an explanatory phrase within the parentheses, thus:

> The functions are stored in a dynamic link library (hereafter referred to as a DLL).

Do not use periods within the acronym or abbreviation.

Table 1.2 lists some acronyms that are known mainly as acronyms and do not need to be spelled out.

Table 1.2 Acronyms and abbreviations without expansions

ASCII	EBCDIC
BASIC	PC
COBOL	CLIM
CPU	NT
LISP (But "Lisp" is preferred. Harlequin uses "Common Lisp".)	

To form the plural of an acronym or abbreviation, just add "s":

> None of the PCs work.

Other acronyms and abbreviations are listed in the Glossary, together with their expansions.

1.2 Capitalization

See Section 1.7 for a description of document elements and product elements to capitalize.

Capitalize proper names. These include:

- **Real world elements:** people (Steele), places (Cambridge), companies (Harlequin), products (Watson)
- **Software elements:** the names of windows (the Error Message window) and buttons (Cancel)

You may need to follow another usage to match the appearance of these elements in the interface. Elements within windows also have names. This can lead to ambiguity as in the following sentence:

> Type a name in the Save as text box and click OK.

Does the reader recognize that "Save as" is the name of the field in the window when "as" does not have an initial capital letter? In this case you should quote the name of the window:

Type a name in the “Save as” text box and click OK.

In captions and section headings Harlequin use sentence style (only an initial cap) rather than headline style (caps for all significant words). Proper names and acronyms should retain their proper capitalization. Book titles and chapters use headline style. For a more complete definition of headline style capitalization, refer to Chicago, paragraph 7.127.

1.3 Citations

Citations of outside texts should consist of a mostly complete bibliographic reference at first mention, followed by an abbreviated form of the book name which should be used consistently throughout the rest of the document.

Table 1.3 Citation conventions

Book	First mention citation (suggested)
AMOP	See *The Art of the Metaobject Protocol* by Gregor Kiczales, published by MIT Press, hereafter referred to as *AMOP*.
CLtL2	See *Common Lisp: The Language*, 2nd Edition, by Guy L. Steele, Jr., published by Digital Press, hereafter referred to as *CLtL2*.
Red Book	See the Red Book, *The PostScript® Language Reference Manual*, 2nd Edition, published by Addison Wesley.
DRM	See *The Dylan Reference Manual*, by Andrew Shalit, published by Addison Wesley Developers Press, hereafter referred to as the *DRM*.

A list of all Harlequin publications can be found in the Doc Group Information database in Spring. See Section 3.1 for details on how to see this database. When referring to a Harlequin publication, use the complete title of the book. For example:

See the *LispWorks User’s Guide* for more information.

1.4 Code examples

Many documents require examples of source code written in an appropriate language (commonly Lisp, C, or ML). To format these examples using FrameMaker, use the Code family of paragraph formats described in Section 2.2.

Make sure you break up segments of source code into meaningful chunks. Segments such as function or procedure definitions, and constructs such as loops, should be separated from each other with white space, to give the appearance of “paragraphs” of code. In FrameMaker, each line of code is a separate paragraph, and specialized paragraph types are used to introduce the extra space between segments, as shown below.

```
Use Code-first for the first line in a segment.
Use Code-body for other lines in a segment.
Use Code-last for the last line in a segment.

Use Code-line for a segment that consists of only one line.
```

Body text is indented more for reference manual entries than for other types of Harlequin document. Therefore, source code examples need to be indented appropriately for reference manual entries as well. If you are formatting a source code example for use in a reference manual, you should use the RCode family of paragraph formats. Use RCode-first instead of Code-first, RCode-body instead of Code-body, RCode-last instead of Code-last, and RCode-line instead of Code-line.

Indent source code using spaces, rather than tabs. Although you can use normal spaces, it is better to use hard spaces. In FrameMaker, you can insert a hard space by typing `Ctrl+Space`. At the time of writing, WebMaker formats code examples that use tabs incorrectly.

By default, all Harlequin document templates have the Smart Spaces feature turned on, preventing you from typing two or more consecutive normal spaces. You do not need to worry about this feature if you use hard spaces in your code examples. However, if you prefer to use normal spaces, you need to turn Smart Spaces off before you can format source code examples in a document. Do this as follows:

1. Choose **Format > Document > Text Options** in the FrameMaker document window.

2. Uncheck the Smart Spaces check-box in the Text Options dialog.

When using quotation marks in source code examples, try to use only the characters `"` (straight double quotes), `’` (forward single quote) and `‘` (backward single quote). In particular, try not to use the `'` (straight single quote) character; most text editors display a single quotation mark as a forward single quote rather than a straight single quote. This is especially important in Common Lisp code, which uses the forward single quote and backward single quote for different purposes. Type **`Ctrl+"`** to type the `"` character when Smart Quotes is turned on. Note that it is best to turn Smart Quotes off if you are doing extensive formatting of source code examples. Do this as follows:

1. Choose **Format > Document > Text Options** in the FrameMaker document window.
2. Uncheck the Smart Quotes check-box in the Text Options dialog.

See Chapter , “Formatting Lisp Code” for details about formatting Common Lisp code in particular.

1.5 Comments

Comments may be added to documents using the Comment conditional text format described in Section 2.6. By convention, comments should begin with the string “COMMENT:” so as to make them easily identifiable in the document text. You may also like to add your name, and possibly the date to all comments you add as well.

The following FrameMaker macro will create such a comment for you. If you would like to use it, copy it into your **`~/fminit/fmMacros`** file, and edit it as appropriate. You need to insert your own name, and, if you wish, bind the macro to some key other than F3 (to which it is bound in the example below).

```
<Macro DatedComment
       <Label Dated Comment>
       <Trigger /F3>
       <TriggerLabel F3>
       <Definition ^4/START_DIALOG comment\s/END_DIALOG
    C/Left \s/Right OMMENT\:\s^0
    /START_DIALOG cuc\s/END_DIALOG \sinsert-your-name-here
    /Left /Left /Left /Left /Left
    /Left /Left +/Right \!sv/START_DIALOG
    ^/Tab /Tab /Tab /Tab \s/END_DIALOG
    /START_DIALOG 1/Return /END_DIALOG
    /START_DIALOG ^/Tab /Tab /Tab /Tab /Tab /Tab /Tab \s
    /END_DIALOG
    /Left \s >
       <Mode NonMath>>
```

To use the macro, type F3 (or whatever key combination you bind it to). The macro adds the string “COMMENT:” at the start of the comment, and the current date and your name at the end. The macro also ensures that comment conditional text is used, and that the cursor is positioned correctly for you to type the text of your comment. When the macro has finished, type the text of your comment.

1.6 Contractions

Contractions are almost never necessary in Harlequin manuals. They may be preferred in advertising and marketing material for a “friendlier” tone, but some studies show they actually slow comprehension.

Table 1.4 Expansions for contractions

Use this expansion …	… instead of this contraction
cannot	can’t
do not	don’t
it is	it’s
you are	you’re

1.7 Conventions

Document conventions aim at consistent typographic signals for common elements.

Bold	Use bold text for the words “Note:” or “Warning:” introducing a paragraph of special text. See Section 1.16.
Button	Use this font for text that refers to menu commands. Use the > character to denote the progression from menu to submenu and down, for example “Choose **File > Utilities > Compare Documents**.” Also, use this font for button names, for example “Click **Cancel**.”
`Code`	Use code text for code references inside body paragraphs, command keywords, key combinations, filenames, directory names.
Italic	Use italic text for book titles, the first mention of a term defined in the glossary, and emphasis. Note, however, that emphasis should be used *very* sparingly in any document.
Variable	Use this font for any mention of a code variable or argument in a body paragraph, and to denote variables and arguments in the syntax line of a reference manual description.

See Section 2.3 for more information on how to use these fonts in FrameMaker itself.

Capitals: mixed case

Use mixed case for proper names, including the names of windows, dialog boxes and options within windows and dialog boxes. For example, “Enter the value in the

	Top Margin text box of the Page Layout window.” Capitals in mixed case for proper names is the standard, but documentation must match the interface. Quote names only when ambiguity is likely. See Section 1.2 for more details.
Initial capitals	In section headings and figure and table captions, capitalize only the first word (sentence style), rather than every significant word (headline style). Use headline style for chapter and book titles. A precise definition of headline style can be found in Chicago, paragraph 7.127.
`Ctrl+C`	This is how to write a key combination when the user must press and hold down one key and then press another.
Capitals for keys	In key combinations, all alphabetic characters are written in upper case; **`Ctrl+X`** is correct, **`Ctrl+x`** is incorrect. Do not use capitalization to indicate a distinction between upper and lower case in key combinations. If the software makes a distinction between key combinations according to the use of the Shift key, write **`Ctrl+Shift+A`** to mean shifted “A” and **`Ctrl+A`** for unshifted “A”.
Alt	Alt key
Ctrl	Control or Ctrl key
Esc	Escape key
Meta	Meta key (Command key on Apple keyboards)
Return	Return key

Shift	Shift key
Space	Space bar
Super	Super key

Note: There is one specific exception regarding key combinations. When specifying a key combination for the EMACS text editor, use the EMACS model for key combinations (C=Control, S=Shift, M=Meta, with a hyphen used to separate characters, rather than a +).

1.8 Copyright

The copyright notice appears on the copyright page, the back of the title page. It has one of the following forms:

Copyright © *year*, *year*, *year* by The Harlequin Group Limited.

Copyright © *year–year* by The Harlequin Group Limited.

If you use the latter form, you must use an en dash, which FrameMaker provides via the keyboard shortcut **`Ctrl+Q Ctrl+P`**.

For example:

Copyright © 1992, 1994 by The Harlequin Group Limited.

Copyright © 1987–1994 by The Harlequin Group Limited.

Using both “Copyright” and the symbol © is redundant but provides some safety if the symbol font is missing.

Caution: Copyrights belong to The Harlequin Group Limited. Notice capitalization of “The” and spelled out “Limited” in this legal use.

1.9 Footnotes

Footnotes can always be avoided in Harlequin documentation. The standard appearance of footnotes in FrameMaker does not match well with the Harlequin templates, and there are known problems with FrameMaker footnotes. Do not use footnotes in Harlequin manuals.

1.10 Hyphenation

Hyphenation is an important factor to consider when using many computer terms. Variants of terms such as online (online, on-line, on line), and of GUI elements such as checkbox (check box, checkbox, check-box) are all in use in the industry. Harlequin usage follows these general guidelines:

- When describing a term that applies to a GUI (either an element such as a check box, or a descriptive term such as double-click), try to use the term given in the glossary of the Windows Interface Guidelines, published by Microsoft. A printed version of this publication should be available at your site.

  Of course, exceptions can be made to this rule if there is good reason (for example, if it contravenes accepted usage for the audience of the document).

  Please refer to “Harlequin spellings” in the Glossary for common examples.

- When using other terms that may or may not need hyphenating, and that are not defined explicitly in a dictionary, tend towards the hyphenated variant.

  The thinking behind this is as follows: As a term is accepted into common use, it tends to move from a two word term, to a hyphenated term, and finally to a single word. By taking the middle option, our documentation does not come across as old fashioned, and yet you do not risk using a single word term inappropriately.

  Please refer to “Harlequin spellings” in the Glossary for common examples.

When considering whether or not to hyphenate a term, you should also remember the grammatical context in which it is being used. In particular, there is no need to use the hyphenated term if it is not being used to modify a noun. Consider the following two examples, both of which are correct:

> The run-time system contains the garbage collector.
>
> The error is only detected at run-time.

Most Harlequin documents are set ragged right and allow FrameMaker to hyphenate words automatically in order to limit how ragged the right margin is. Hyphenation rules are generally those of the dictionary and the FrameMaker dictionary.

To prevent hyphenation of a word that FrameMaker would have automatically hyphenated, type **`Shift+Meta+Hyphen`** or **`Esc N S`** before the word. This forces the word onto the next line if it would break at the end of a line. This can be important when command words that contain hyphens might become ambiguous because of hyphens inserted by FrameMaker. For example, a FrameMaker-inserted hyphen could make the LispWorks **`ADD-SPECIAL-FREE-ACTION`** symbol confusing by adding a hyphen and breaking the word after **`SPEC`** ("**`ADD-SPEC-IAL-FREE-ACTION`**"). Suppressing hyphens with **`Shift+Meta+Hyphen`** still allows the command word to break at any of the hyphens.

Code examples are not safely left for automatic hyphenation. Hyphenation should be turned off in code and line breaks should be handled explicitly. See Appendix A, "Formatting Lisp Code" for information on formatting Lisp code in general.

1.11 Indexes

Indexes should be built with the template **`IX.doc`** (see Section 2.1). They are two-level, two-column indexes.

Refer to a number of indexes as "indexes", not "indices".

Use markers of type Index to enter text for index entries.

1.11.1 "See" and "See also"

See and *See also* index entries use italics and no page number.

For example, the following entry refers the reader to another term in the index:

> backscreen. *See* previous screen

The FrameMaker index marker to produce this is

```
<$nopage>backscreen. <Italic>See<Default Para Font> previous
screen
```

For *See Also* entries, you want to control Frame's sorting to put the entry at the end of all the subtopics. This marker text:

```
<$nopage>universe definition:<Italic>See also<Default Para Font>
universe counts[universe definition:zzz]
```

results in an index entry like:

universe definition

  entering new selection criteria 167
  using queries 33
  See also universe counts

1.11.2 Denoting principal references in the index

Use bold type to denote principal references in an index entry. For example:

treeshaking 34, 45, **65**, 95

Always include a key on the first page of the index explaining this. For example:

> Page numbers shown in bold typeface denote the principal entry for that item.

Note: As of November 1996, such keys should only appear in the paper version of any manual, not the HTML version. Make sure that this key has the Paper conditional text format applied.

If there is only one locator for an index entry, do not use bold type to denote it as a principal reference.

To make the page number come out in bold, set the character format to Bold at the end of the marker text. FrameMaker will preserve this change and print the page number in bold as intended. This marker:

```
treeshaking<Bold>
```

will appear with "treeshaking" in Palatino, and the page number in Palatino Bold.

You can do this even if the marker contains text for more than one index entry, separated by a semicolon. The following marker text:

```
treeshaking<Bold>;trap
```

Produces entries like these:

treeshaking	**65**
trap	65

If your marker controls entry sorting, make sure that the character formatting information appears before the sorting information, like this:

```
<Italic>1989<Default Para Font><Bold>[nineteen eighty nine]
```

Do not follow Frame's other recommended solution to this problem, of using a different marker type for the principal entries. See *Using FrameMaker* for more details.

Note: As of November 1996, entries displayed in bold text in a FrameMaker index do not appear bold in the HTML index after it has been run through WebMaker.

1.11.3 Sorting index entries in multiple places

Index entries that begin with special characters such as ":", "*", and so on should be indexed under both the special character and the words that follow. For example, the entry for **`:width`** should be indexed under both ":" and "width". The index marker text would appear like this:

```
<Code>\:width<Default Para Font>[\:width;width]
```

The index marker for **`*pprint-length*`** would look like this:

```
<Code>*pprint-length*<Default Para Font>[*pprint-length*;pprint-
length*]
```

See *Using FrameMaker* for more details.

1.12 Part numbers

Part numbers need to be generated for each manual or document that the doc group releases. The part number changes with each new external release of a document (or, in the case of internal-only documents such as this manual, each "significant" release).

Part numbers should follow the structure shown below:

Product–Version–Manual–Platform–Edition

Product is a 2, 3, or 4 letter abbreviation denoting the product for which this is a manual. Abbreviations already in use include LW (LispWorks), LCL (Liquid Common Lisp), SW (ScriptWorks), W (Watson), (HCIS) HCIS, WM (Web-Maker).

Version should be the complete number of the software release to which this manual applies. The part number is the only place in which the exact version number of the software is recorded, and it should therefore mirror the actual software as closely as possible. Include periods (.) that are present in the actual version number, and append ".B" for a beta release, ".B2" for a second beta release, and so on.

Manual should be a short abbreviation denoting the name or type of the manual. Commonly used abbreviations are UG (User Guide), RM (Reference Manual), RN (Release Notes), and IN (Installation Notes).

Platform is an optional field that denotes the platform to which the manual applies, in cases where there are several platform-specific versions of the same manual. If the manual you are working on is not platform-specific, you may omit this field. There are three possible values: PC (PC) M (Macintosh), and U (UNIX).

Edition is an optional field that denotes the edition of the manual. This is to allow for the rare case when we issue two versions of a manual for precisely the same software release. The first edition of any manual is 1, and the field is not required in this case.

En dashes should be used to separate each field in the part number. In FrameMaker, you can create an en dash using the keyboard combination **`Ctrl+Q Ctrl+P`**.

This system may seem somewhat confusing if you have not generated many part numbers. The bottom line is that we strive for uniqueness; any part number should uniquely identify the particular manual it is the part number for. If, in generating a unique part number, you do not stick rigidly to the structure described here, no one will shoot you.

1.12.1 Some examples

Here are some examples of correctly formed part numbers; some real, and some imaginary:

`SW-4.0.11-OEM-U-2`	ScriptWorks 4.0.11 *OEM Manual* for UNIX, 2nd edition.
`HCIS-1.0.B-GEN-PC`	HCIS 1.0 Beta *General Information Manual* for PC.
`LCL-5.0.B-RN`	Liquid Common Lisp 5.0 Beta *Release Notes*.
`CLW-1.0.B-UG`	Common LispWorks 1.0 Beta *User Guide*.
`LW-4.0r2-RM-PC`	LispWorks 4.0 Revision 2 *Reference Manual* for PC.

1.13 Pronoun agreement

Use plural pronouns to agree with plural nouns and either second-person or singular pronouns to agree with singular nouns.

Avoid constructs like “the user … they ….” First choices are “the users … they” or “when you”. A distant second is to face the gender issue head on: “the user … he / she.”

1.14 Punctuation

This section discusses the use of punctuation in Harlequin documentation.

1.14.1 Commas

Commas can appear in these cases:

- after introductory elements
- between items in a series
- between main clauses of a compound sentence

- setting off parenthetical elements

Restrictive clauses should not be set off with commas.

1.14.2 Ellipsis / marks of omission

An ellipsis should be used to mark omission.

To improve spacing and satisfy the Spelling Checker, use the FrameMaker ellipsis character (…) by typing **`Ctrl+Q Shift+I`** rather than three periods (...) or three spaced periods (. . .). Enter a space before the ellipsis.

Omission at the end of a sentence requires a period following the ellipsis character.

1.14.3 Fonts for punctuation

The font for marks of punctuation at the end of a font shift is the default paragraph font. For example, Choose **File > Save**. The Button font is used for the menu sequence, but the sentence concludes with a Palatino period that matches the paragraph default.

1.14.4 Periods / full stops

Periods (full stops) should appear in these cases:

- At the end of list elements that form complete sentences.
- After numerals in numbered lists.

1.14.5 Quotation marks

Quotation marks should be used as rarely as clarity allows.

Most proper names of interface parts (buttons, menus, and so forth) are distinguished by font or capitalization and do not require quotation marks. See Section 1.7.

- Standard *Chicago* use is double quotation marks. This convention differs in American and British use, and we follow *Chicago*.

- Quotations within quotations are marked with single quotation marks, as specified in *Chicago*. This reverses the British order of nesting. Quotations within quotations should almost never occur in technical documentation.
- Punctuation within quotation marks should match the original being quoted.

  This is the British rather than American standard. American usage includes commas and periods within the quotation marks regardless of the original. This can lead to trouble in computer and other technical situations. Instructions like the following make clear that the user does not enter a period:

  Press “k”.

  Chicago follows American usage with reservation and recognizes that some American publishers follow British usage.

1.15 Release-specific information

The title page should specify the release number, and the copyright page should specify the release number and date (in *Month Year* format) that the documentation describes. In addition, a part number should be specified on the copyright page. See Section 1.12 for details on how to generate part numbers.

The body of a manual should, whenever possible, avoid release-specific references. Differences between this release and earlier release or differences from anticipated releases should be limited to release notes. There are exceptions to this, however, in cases where release-specific information is integral to the content of a manual. The ScriptWorks Plugin Kit is one such example.

1.16 Special text

Paragraphs of special text that interrupt the normal flow of the document in order to provide important or useful information should be preceded by “Note:” or “Warning:” as appropriate. In both cases the word should be set in bold face, for example:

> **Note:** Opening a file causes a lock file to be created in `/tmp`.

Use “Warning:” for anything that might cause the user problems and “Note:” for salient points that you want to bring to the reader’s attention for some other reason.

1.17 Table of contents

The table of contents should be built with the template **`TOC.doc`** (see Section 2.1). You must use the **Set Up File** command in the book file to specify that the following paragraph types should be included in the TOC:

> Chapter, 1Heading, Preface, Glossary, Appendix, XIndex

Note: If your document does not contain one or more of these, they will not be available as choices in the Set Up Table of Contents dialog. That is OK; simply add the ones that are available.

1.18 Tense

Documentation can usually be written in the simple present tense. Rewrite sentences like the following:

> When using this mode, if the printer runs so fast that it catches up with the RIP, a page buffer will be created and the current page will be ripped into it.

Write this instead:

> If the printer runs so fast that it catches up with the RIP, ScriptWorks creates a page buffer and writes the current page into it.

Sentences describing capability should use “can” rather than “will”.

1.19 Trademarks

Trademarks are marked on the first occurrence only, usually on the title page or cover, in the form product® or product™. For example:

> ScriptWorks®
>
> Harlequin Precision Screening™

Use **`Ctrl+Q (`** to get the registered trademark symbol (®). Use **`Ctrl+Q *`** to get the trademark symbol (™). To improve their appearance on the printed page, you should make the ® and ™ characters superscript. On title pages, you may also have to experiment with the size of the characters to achieve the best effect.

Trademarks are acknowledged on the copyright page. More information about trademarks can be found in the Glossary.

1.19.1 Harlequin trademarks

All the trademarks that Harlequin owns in a product line should be claimed on the copyright page of each manual for products in that line. So each ScriptWorks book lists all the ScriptWorks trademarks.

Trademark notices are kept current on sample copyright pages for each product in the **`DOCshare-template`** compound in HOPE. See Section 2.1.

1.19.2 Other companies' trademarks

Harlequin documents do not attempt to identify all the trademarks and their holders in our documentation. The following statement concludes the copyright page trademark notice:

> Other brand or product names are registered trademarks or trademarks of their respective holders.

Exceptions are listed in the next sections.

1.19.2.1 Adobe

Adobe trademarks are noted in accordance with the *Adobe Trademark Reference Manual.* So the first use of PostScript is marked with the ® symbol. The copyright page trademark notice says PostScript is a registered trademark of Adobe Systems Incorporated. Any other Adobe products should receive the same treatment. This notice belongs in all the ScriptWorks manuals and may belong in manuals for the other Harlequin products. For example:

> Adobe, Adobe Photoshop, Adobe Type Manager, Display PostScript and PostScript are registered trademarks of Adobe Systems Incorporated.

In addition, Adobe has formally requested that Harlequin observe other conventions when Harlequin refers to its trademarks:

- Do not reproduce trademarked Adobe graphics, like the Lady Golfer image.
- The word “PostScript” refers to the PostScript family of products owned or endorsed by Adobe. (ScriptWorks is not among them.) The page description language itself is called “the PostScript language”.
- When using “PostScript” in reference to products or entities that do not originate from, or are not endorsed by, Adobe (the usual case for Harlequin), use the formulation “PostScript language” or “PostScript language-compatible”.

  For example, we say that ScriptWorks is “fully PostScript language-compatible” rather than “fully PostScript compatible”. We say that our utilities “manipulate the PostScript language files produced by other applications” rather than “manipulate the PostScript produced by other applications”.
- ScriptWorks (and other non-Adobe RIPs) should be referred to as a “PostScript language-compatible interpreter”.

1.19.2.2 HP

HP has asked that we treat their trademark operating system name as “the HP-UX operating system” rather than as simply HP-UX and we have agreed in the Lucid contract. This does not require notice on the copyright page but determines our style in mentioning this product.

1.19.2.3 Contractual obligations

We may have obligated Harlequin to advertise the copyright or trademark of a product, perhaps one included in a Harlequin product.

1.19.3 Disclaimers

The following disclaimer should be used on all copyright pages:

> The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by Harlequin Limited, Harlequin Incorporated, Harlequin Australia Pty. Limited, or The Harlequin Group Limited. The Harlequin

> Group Limited assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

1.19.4 U.S. Government "Restricted Rights"

Harlequin software documentation requires a paragraph about U.S. Government rights. This appears on the copyright page:

> US Government Use
>
> The *xxx* Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be `unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

The *xxx* should be replaced with the name of the software product.

1.19.5 Remaining Issues

The Legal group is responsible for the following information:

- List of Harlequin trademarks and registered trademarks.
- List of Adobe trademarks and style manual from Adobe.
- List of any other trademark and copyright obligations.

1.20 US English

Harlequin uses US English spelling for most words. Usually this is the first spelling listed in the *Random House Webster's College Dictionary.* As a general rule of thumb, US English spellings tend to be shorter than UK English spellings. Table 1.5 identifies the following most common differences between US and UK spellings (adapted mostly from *The Oxford Companion to the English Language*, OUP, 1992, edited by Tom McArthur):

Table 1.5 Common differences in US and UK English

Group	Notes	Examples (US / UK)
colo(u)r	The *u* in UK English *–our* can always be dropped from the US English.	color / colour favor / favour labor / labour
centre / center	UK English *–re* forms are changed to *–er* for US English, except where this leads to an implied change in pronunciation (hence, no *acer, mediocer, massacer* …).	center / centre fiber / fibre liter / litre
–ize / –ise	In general, if a verb can take both *–ise* and *-ize* forms, *–ize* is preferred in US English. Verbs that can take both usually form nouns in *–tion*, and verbs that only take *–ise* do not. There are exceptions though.	customize / customise organize / organise
–lyze / –lyse	In all cases, the UK English *–lyse* translates to US English *–lyze*.	analyze / analyse paralyze / paralyse
–ice / –ise	Where, in UK English, *–ice* is used for a noun, and *–ise* for the corresponding verb, only *–ice* is used in US English, unless the pronunciation of the word changes depending on its use.	practice / practise (verb) *but* devise / devise (verb) device / device (noun)
–nce / –nse	Where, in UK English, *–nse* is used to denote a verb form, and *–nce* a noun form, only *-nse* is used in US English	license / licence (noun) offense / offence (noun)
–st	In US English, variants of prepositions and conjunctions that end in *–st* are seldom used.	among / amongst while / whilst amid / amidst
instil(l)	If a UK English word has a single written vowel plus *–l*, the US English equivalent has the vowel plus *–ll*. The exception to this is *annul*, which is identical in both US and UK forms.	instill / instil fulfill / fulfil enroll / enrol

Table 1.5 Common differences in US and UK English

Group	Notes	Examples (US / UK)
final –l(l)	In US English, verbs that end with a single written vowel followed by *–l* or *–ll* keep them before *–s* and *–ment*. Before a suffix beginning with a vowel, *–ll* words keep *–ll*, *-l* words should generally follow the general rules for doubling final consonants. This is a horrible rule, which you should probably deal with on a per-case basis!	traveling / travelling installment / instalment
–og(ue)	US English sometimes drops *ue* in *–ogue*, although only *catalog* is in wide use. At Harlequin, *dialog* is exclusively used when referring to dialog boxes in a GUI.	catalog / catalogue dialog / dialogue
(o)estrogen	In US English, *e* is more common than *oe*, which need never be used in our documentation. Note that it is unlikely that this issue will arise frequently in our documentation anyway.	estrogen / oestrogen homeopathy / homoeopathy
(a)esthete	In US English, *e* is more common than *æ*, except in the form *aer*, which remains the same as UK English. Hence *aerate*, *aerobics*, *aerosol*.	esthete / æsthete eon / æon encyclopedia / encyclopædia

The differences between UK and US English can be complex and subtle, and involve not just spelling, but grammar and word usage as well. The list in Table 1.5 is by no means complete. See the Glossary for a list of common US spellings that you might encounter. As mentioned above, the *Random House Webster's College Dictionary* is probably the most convenient spelling authority for individual cases. For subtle points of usage that you cannot resolve, ask the US members of the Documentation group!

1.21 Voice

Use active rather than passive constructions when possible. This advice is cliche but still not sufficiently followed. For example, use this:

ScriptWorks automatically loads fonts whenever a job requires them.

not this:

Fonts are loaded into ScriptWorks automatically whenever they are required.

2 Formats

Harlequin documentation uses a number of book designs and sets of formats in FrameMaker to lay out the documents in accordance with those designs. In FrameMaker these are called document formats, paragraph formats and character formats. This chapter describes the templates and formats of the current design. It lists the formats and describes their uses to make the tools easy to learn.

The templates for this design are available in HOPE in the **`DOCshare-template`** compound. Each type of file in a book (title page, copyright page, preface, chapter, appendix, and so on) has its own template file there. See Table 2.1 for a listing of each template file and its typical usage. All of these files are automatically checked out into **`~doc/share/template/`** every night at each Harlequin site where members of the documentation group are based.

2.1 Document format

The document format proper defines continuous page numbering for the entire book, automatic left and right quotes, a limit on a single space between words, and some other document-wide features.

Most of the overall document design is set by Frame's master pages. These are included with the template documents for this design. The master pages define a 7 1/2 by 9 inch page with headers and footers.

You have to use a FrameMaker book and update it to get continuous page numbering for multiple files. Numbering starts with an unprinted 1 for the title page through lower case Roman numerals for the front matter (copyright, table of contents, and preface). Numbering then starts again with 1 in Arabic numerals for the first page of chapter 1 and proceeds through the index to the end of the book.

The document, paragraph, and character formats are available when you copy the template documents or their formats into your document.

Table 2.1 shows the several template files and their standard uses.

Table 2.1 Harlequin document template files

File	Usage
`Title.doc`	Title Page
`Copyrght.doc`	Copyright page
`Lwcprght.doc`	Copyright page for LispWorks manuals
`Swcprght.doc`	Copyright page for ScriptWorks manuals
`TOC.doc`	Table of contents
`Preface.doc`	Preface and glossary (minor differences are documented in the template file)
`Chapter.doc`	Document chapters
`Chapter-ref.doc`	Document chapters with reference material
`Appendix.doc`	Appendix
`IX.doc`	Index
`Short.doc`	Short documents: release notes, white papers, and so on
`ShortTOC.doc`	Table of contents for use with **`Short.doc`**

Note: Ultimately, the aim is to replace **`Chapter.doc`** with **`Chapter-ref.doc`**. At the time of writing, however, **`Chapter-ref.doc`** is still being tested, and so the two templates are being maintained in parallel. If any of the files in a book contain reference material, you should use **`Chapter-ref.doc`** for all of the chapters in that book. If there is no reference material, you can use **`Chapter.doc`**.

The regular title, copyright, appendix, and index templates are compatible with the **`Short.doc`** and **`ShortTOC.doc`** templates.

See *Using FrameMaker* for more information on document formatting.

2.2 Paragraph formats

Writers choose the formats of paragraphs that match the content of the material and produce the layout they need. It is important to use the set of standard paragraph formats in their standard form. FrameMaker allows easy modification, but fine tuning in this way makes the inevitable future reformatting for a new design or new system fail.

The following alphabetical list describes the purposes of each paragraph format. Some formats appear only in particular templates; for instance, the 1Heading-App format appears only in **`Appendix.doc`**.

1Heading	This is a top level heading. It is used for the major sections of a chapter, as in “Paragraph formats” above. It has autonumbers.
2Heading	This is the second level heading. It has autonumbers.
3Heading	This is the third level heading. It has autonumbers.
1Heading-App	This is the top level heading in appendices. It is used for the major sections of an appendix. It does not appear in the table of contents. It has autonumbers.
2Heading-App	This is the second level heading in appendices. This does not appear in the table of contents. It has autonumbering.

3Heading-App	This is the third level heading in appendices. This does not appear in the table of contents. It has autonumbering.
Appendix	This formats the appendix title that appears at the top of the first page of an appendix.
Appendix-Number	This formats the appendix number that appears at the top of the first page of an appendix.
Body	This is the format of body paragraphs, the most common element in most documents. The introductory paragraph of this section is a Body paragraph.
Body-Indent	This is just like Body, but indented. It is used as the text following Interface-type.
Body-Next	This is a specialized version of the Body paragraph used when you want to require that the next element stay on the same page as the Body-Next paragraph. It is commonly used when you introduce a list with a colon. Body-Next keeps an introduction like “See the following sections:” with the first item in the list that follows.
Booktitle	This produces the large font title on the title page of the book. The Booktitle can be a product name or the title of the book. The Booktitle can be 1, 2, or 3 lines long.
Bullet	This produces an indented list item beginning with a bullet.
Bullet-C	This continues the text under a bullet with a new paragraph. It produces a paragraph with the same indentation for text as Bullet but without the bullet.
Callout	This formats text used with figures in Helvetica.

CellBody-small	This formats the table entries when a smaller type face is required. In other cases, use Body.
CellHeading	This formats the table column headings.
Chapter	This formats the chapter title that appears at the top of the first page of a chapter.
Chapter-Number	This formats the chapter number that appears at the top of the first page of a chapter.
Code-body	This formats the middle lines of a fragment of code, program input, or program output, in Courier. See Section 1.4 for details about how to use this format.
Code-first	This formats the first line of a fragment of code, program input, or program output, in Courier. See Section 1.4 for details about how to use this format.
Code-last	This formats the last line of a fragment of code, program input, or program output, in Courier. See Section 1.4 for details about how to use this format.
Code-line	This formats a single line of code, program input, or program output, in Courier. See Section 1.4 for details about how to use this format.
Contents	This formats the in-chapter table of contents that appears as the first paragraph in a Watson chapter.
Copyright-title	This format is used for the text “Copyright and Trademarks” at the top of a copyright page. Since this text is already present in the appropriate template files, you rarely need to use this format yourself.
Copyrighttext	This formats the text on the copyright page in a small point size.

Copyrighttext-small	
	This formats the text on the copyright page in a very small point size. Use this to force all copyright material onto a single page when necessary.
Description	This formats the common two-column tabular material, such as this description of each paragraph format. The item being described is followed by a tab and indented descriptive text.
Description-C	This formats continuation of Description in a following paragraph. It uses the same indentation as the descriptive text in Description without the initial line beginning farther to the left.
Description-Item	
	This formats two-column material like Description and is used when the text of the left column is wider than the left column. It is followed by Description-C for the right column of text. The definition you are reading is a Description-Item followed by Description-C.
Edition	This formats the edition information on the title page.
Exec-summary	Used in the Title page only, this enables you to add a description of the purpose or content of the book. It is only useful for documents intended for publication on the WWW and you should apply the WWW conditional text tag to the paragraph. Text in this paragraph appears before the top-level links in the final HTML, after running the manual through WebMaker.
Figure	This formats the automatically numbered caption for a figure.
Glossary	This is the title of a glossary.

Glossary-definition	This formats a definition of a glossary entry.
Glossary-entry	This formats a term in the glossary. It is followed by Glossary-definition for the text defining the entry.
In-line-interface	This formats the name of a function, variable, keyword, or other programming language item in an unobtrusive style, ranged with the normal margins. Use it only for brief discussions of the named item — when writing complete reference descriptions, turn to **`Chapter-ref.doc`** and the REntry family of paragraphs. When you hit Return, the In-line-interface paragraph is followed by an In-line-interface-type paragraph on the same line.
In-line-interface-type	This formats the type of an interface in italic, aligned at the right margin. Examples of interface-type are: Function, Generic Function, Class, Variable. This is followed by Body-Indent.
Preface	This is the title of a preface.
Pref-Heading	This formats an unnumbered heading in a preface.
Pubcities	This formats the list of cities where Harlequin has offices, under the logo, on the master page of the book title page.

A section of formats whose names begin with R follows. These formats are used for writing full reference entries in chapters based on **`Chapter-ref.doc`**. Most of the formats are special versions of standard formats like Body, Description, and so on, that provide slightly more indentation from the left margin. This is necessary because our formats for setting up full reference entries (REntry, REntry-head and REntry-type) make the main text column narrower. Note that R-versions of paragraphs are rarely indented further than the indented main text column. Thus, RCode-line is not indented any further than RBody. Contrast this with Body and Code-line. See **`Chapter-ref.doc`** for more details.

RBody	**`Chapter-ref.doc`** only. The reference entry version of Body.
RBullet	**`Chapter-ref.doc`** only. The reference entry version of Bullet.
RCode-body	**`Chapter-ref.doc`** only. The reference entry version of Code-body. See Section 1.4 for details about how to use this format.
RCode-first	**`Chapter-ref.doc`** only. The reference entry version of Code-first. See Section 1.4 for details about how to use this format.
RCode-last	**`Chapter-ref.doc`** only. The reference entry version of Code-last. See Section 1.4 for details about how to use this format.
RCode-line	**`Chapter-ref.doc`** only. The reference entry version of Code-line. See Section 1.4 for details about how to use this format.
RDescription	**`Chapter-ref.doc`** only. The reference entry version of Description.
RDescription-C	**`Chapter-ref.doc`** only. The reference entry version of Description-C.
RDescription-Item	**`Chapter-ref.doc`** only. The reference entry version of Description-Item.

REntry	**`Chapter-ref.doc`** only. This formats the name of a function, variable, keyword, or other programming language entity in a reference style. When you hit Return, the REntry paragraph is followed by an REntry-type paragraph on the same line.
REntry2	**`Chapter-ref.doc`** only. This is exactly the same as REntry, but is required for use under a 2Heading. If you have a reference entry that comes in the context of a 2Heading, use this paragraph rather than REntry.
REntry-head	**`Chapter-ref.doc`** only. This formats headings under a reference entry. Examples of heading text are Summary, Library, Module, Arguments, and Values. This is followed by RBody.
REntry-type	This formats the type of an entry in italic, aligned at the right margin. Examples of entry-type are: Function, Generic Function, Class, Variable. This is followed by REntry-head.
RStep	**`Chapter-ref.doc`** only. The reference entry version of Step.
RStep-1	**`Chapter-ref.doc`** only. The reference entry version of Step-1.
RStep-C	**`Chapter-ref.doc`** only. The reference entry version of Step-C.
RSyntax	**`Chapter-ref.doc`** only. This paragraph should be used instead of RCode-line if the immediately preceding paragraph is an REntry-head. This is needed to improve the look of the HTML document after running through WebMaker.

Step	This formats a numbered list item after the first. The text is indented under the automatically generated number.
Step-1	This formats the first item (1) in a numbered list. This format is necessary to restart a numbering stream. Otherwise it is identical to Step.
Step-C	This formats continuation of Step or Step-1 in a following paragraph. It uses the same indentation as the descriptive text in Step or Step-1 without the initial line beginning farther to the left with a number.
Subtitle	This formats a subtitle on the title page of a book.
Table	This formats the automatically numbered caption for a table.
WWW-TOC	Used only on the Title page, and for documents intended to be put on the WWW. This paragraph triggers an externally generated TOC. The text should be the word “Contents”, and the WWW conditional text tag should be applied to the paragraph.

Paragraph formats whose names begin with X are not generally used by writers. They are used on master pages in template design.

XAbove2LineTitle	This is the empty first paragraph of a two-line Booktitle.
XChapterFooter	This formats the footer of the Chapter master page.
XIndex	This is the title of an index.
XLeftFooter	This formats the footer of the Left master page.
XRightFooter	This formats the footer of the Right master page.

2.3 Character formats

Bold	This formats text in a bold weight without changing its family or size. See Section 1.7 for a description of elements displayed in this format.
Button	This formats text in Helvetica bold in a slightly smaller size. It is used for menu selection sequences and buttons on user interfaces. Selection sequences look like this: “Choose **Format > Document**.” See Section 1.7 for a description of elements displayed in this format.
Callout	This formats text used to label elements in a figure.
Code	This formats text in Courier in a slightly smaller size. It is used to represent text that can be read or displayed by machine. See Section 1.7 for a description of elements displayed in this format.
Italic	This formats text in an italic style without changing its family or size. See Section 1.7 for a description of elements displayed in this format.
Variable	This formats body text in an Italic style. It does not change the size of the text, but it does force the font family to Palatino, to enable its use in Code-based paragraphs. See Section 1.7 for a description of elements displayed in this format.
XWhite	This formats text to display white-on-white. It is used to make necessary FrameMaker elements invisible in one special case. Writers never need to use it.

2.4 Table formats

Table titles appear above the table. They are autonumbered.

No row rules	This formats a table without rules between rows.

Row rules	This formats a table with rules between rows. Use rules for more difficult to read material.

2.5 Cross-reference formats

All document templates contain the cross-reference formats described in Table 2.2.

In general, use formats that include a page number (such as Section & Page) when referring to something which is remote from the current page (that is, at least 3 pages away). When referring to something local, there is less need to quote the page number.

When referring to sections within a document, you should use Section & Page and Section in preference to Heading & Page and Heading. The latter two are provided for occasional situations where they may be necessary (specifically, documents which contain section headings that are not numbered, though our standard templates do not contain such headings).

Table 2.2 Cross-reference formats in document templates

Format Name	Example
Appendix & Title	Appendix B, “The Schema”
Chapter	Chapter 2
Chapter & Title	Chapter 2, “Formats”
Figure Number	Figure 2.1 / Table 3.2
Figure Number & Page	Figure 2.1, page 35 / Table 4.1, page 67
Function	**`add-special-free-action`**
Function & Page	**`add-special-free-action`**, page 35
Heading	“Document format”
Heading & Page	“Document format” on page 25
Number Only	2.1
Page	25
Section	Section 2.1
Section & Page	Section 2.1 on page 25
Title Only	*Document format*

2.5.1 Cross-references in documents intended for the WWW

In principle, documents published on the WWW should never have “& Page” style cross-references, since the concept of a page number in an HTML document is meaningless. Because of time constraints, it is not current doc group practice to create different sets of cross-references for printed and online use, though one can envisage a system which uses the WWW and Paper conditional text formats described in Section 2.6. However, some documents are intended primarily for online use, rather than print, and you should strive to avoid cross-references that include a page number in these documents. This manual is an example of one such document, and the *WebMaker User Guide* is another.

2.6 Conditional text formats

All document templates contain the following conditional text formats:

Comment	Use this format for internal comments that are intended only for other Harlequin employees reviewing the document. You must remember to hide this format before producing documentation to be released externally, in any format.
Doc-Status	This format controls the presence of the word "DRAFT" in the footers of every page in the printed copy of a document. You need never use this format explicitly, but you must remember to hide it before releasing documents in their final (full release) form. **Note:** The format should be shown for beta or internal releases of any document.
Paper	Use this format for any text that should only be visible in the paper version of a document. Remember to hide this format before running any manual through WebMaker.
WWW	Use this format for any text that should only be visible in the HTML version of a document. Remember to hide this format before producing the printed version of any manual.

3 Procedures

This chapter discusses several standard procedures for dealing with documents and methods of document management such as Spring and HOPE.

3.1 Using Spring

Spring is Harlequin's corporate database. It is actually an internal name for Lotus Notes, a database / groupware package of some note (!), and is used to store information about employees, machines, software change requests, and just about anything else. Even the odd functional specification has been known to find its way into a Spring database on some occasions, but this is a closely guarded secret.

Notes runs on UNIX, Windows (all flavors) and the Macintosh. Members of the doc group are most likely to encounter it in its UNIX incarnation, but if you have access to a PC, you are strongly recommended to run it under Windows, since it is much nicer to use on that platform. The currently supported version of Notes is 3.3, but there are rumors that 4.0 is going to be available soon. 4.0 is, apparently, a big improvement over 3.3.

3.1.1 Adding databases to your workspace

When you start Spring, a workspace window appears. This window is used to give you quick access to the databases you use most frequently. It has the appearance of a card index, complete with tabs, so that you can organize your databases according to the way you work. To access databases from the workspace, you need to add them to the workspace. Once added, a database is represented by an icon in the workspace window; you can open any database in your workspace by double-clicking on it. See Section 3.1.2 for a list of useful database you might want to add to your workspace.

You can add databases to your workspace in one of two ways. The most basic procedure is as follows:

1. Choose **File > Open Database** from the Notes menu bar.

   The Open Database dialog appears.

2. From the list of servers, double-click the server local to your Harlequin site. (You may have to supply your password, if you have just started Spring.)

   For instance, if you are based at Hall, choose **`notescam/Harlequin`**. If you are based at 2QC, choose **`notesman/Harlequin`**.

3. Select the database you want to add from the list at the bottom of the dialog.

   This list is similar to a list of files. Items in mixed case represent database names, and items in upper case and surrounded by square brackets [] are directories that contain further databases or directories. Double-click a directory to display its contents.

4. Click **Add Icon** to add the directory to your workspace.

5. Add further databases as you wish.

6. Click **Done** when you have finished adding databases to your workspace.

You can also add databases directly from the Database Catalogue that is maintained at your local server. This catalogue is among the first databases listed in the Open Database dialog when you are connected to your local server as described above, and you may find it useful to add it to your workspace so that you can refer to it easily. The Database Catalogue keeps a list of all the databases available on your local Notes server.

Use the Database Catalogue to add databases to your workspace as follows:

1. Double-click the Database Catalogue icon in your workspace to open it (assuming you have added it to your workspace).
2. Mark any database you want to add to your workspace by clicking in the gray margin just to the left of the database name. A tick appears next to the database name.
3. Choose **Tools > Run Macros > Add Selected Databases** to add all the databases you have selected to your workspace.

3.1.2 Useful databases to know about

This section lists a number of databases that you might want to add to your workspace. Note that some of these databases might be added to your workspace by default when you first run Spring. This list is by no means exhaustive; feel free to add any other databases that you think might be useful to other members of the doc group.

Table 3.1 List of useful databases

Database name	Description	Location (in Open Database dialog)
Database Catalogue	List of all databases.	Top level
Doc Group Information	General database for the doc group	`[PROJECTS]\[DOC]`
Harlequin Information	General information	`[BASICS]`
Harlequin Library	List of technical books and journals	`[BASICS]`
Harlequin Names	List of harlequin employees	`[BASICS]`
Harlequin News	Copy of important internal mails	`[BASICS]`
HIS Call Tracking	List of outstanding calls to sysadmins	`[HIS]\[SYSTEMS]`
Hope Notebook	Misc. info about HOPE.	`[HIS]\[HOPE]`

Table 3.1 List of useful databases

Database name	Description	Location (in Open Database dialog)
Spring User Information	Introduction to using Spring.	**`[BASICS]`**
Systems Notebook	Misc. info from sysadmins.	**`[HIS]\[SYSTEMS]`**

3.1.3 Converting FrameMaker documents for use in Spring

This section shows how to take a FrameMaker document and convert it into RTF format for the Spring database.

1. Save the FrameMaker document as an RTF file using **File > Save As** from within FrameMaker. Give this file the extension **`.rtf`**.
2. Open the Spring database. Position the cursor between quotation marks in the Description field. Choose **File > Import** and give the name of this **`.rtf`** file.

There is also a command line filter that you can use to create RTF files. Save your FrameMaker document as a MIF file, and then run the command

```
>/usr/local/frame/bin/miftortf source.mif target.rtf
```

substituting the name of the MIF file you create and the RTF file you want to create as appropriate.

Note: The creation of RTF files from FrameMaker files is not perfect. Graphics and imported figures in documents almost always fail to convert, and sometimes the conversion process may fail altogether. If you have problems saving using the **Save As** command, then it may be worth trying the **`miftortf`** filter, but do not place too much hope in this, since it is possible that Frame itself uses the same filter when creating RTF files.

3.1.4 Converting Spring documents for use in FrameMaker

Documents in Spring databases can also be imported into FrameMaker, by saving an RTF version of the document. Do this as follows:

1. Choose **File > Export** in Spring.

2. Choose “MicrosoftWord RTF” in the “Save File as Type” drop-down list box.
3. Specify a filename and directory to save the RTF file to. Use the suffix `.rtf` for the filename.
4. Click **Export**.

Now import the RTF file into FrameMaker as follows:

1. Choose **File > Import > File** from the FrameMaker document window.
2. In the Import File dialog, choose the file that you want to import.
3. Make sure that “Copy into Document” is selected (unless, of course, you want to import the file by reference, but this is unlikely for an RTF file).
4. Click **Import**.

   The Unknown File Type dialog appears, and FrameMaker should recognize that it is importing an RTF file and select RTF from the list of file types in this dialog.
5. Click **Convert**.

The RTF file is converted and imported into the current document.

Warning: FrameMaker’s success rate for importing RTF files is variable. Sometimes it will fail to import the file successfully. Sometimes it will cause FrameMaker to crash completely! You are not likely to have much success with anything containing images. *Make sure you save all your files before importing any RTF file.*

3.2 Using HOPE

HOPE is Harlequin’s internal, multi-site, multi-platform source control system. It supports revision tracking, logical groupings of files, checkpointing, and branching. The documentation group uses HOPE to manage Harlequin internal and external documentation, as well as the scripts described in this section.

The documentation group utilizes three scripts which make interactions with HOPE easier.

The **doc_add** script allows you to add documentation units to a given HOPE compound. Its arguments are the same as for the HOPE **add** command, except you can use UNIX wildcards to specify filenames.

The scripts **doc_checkout** and **doc_checkin** can be used to check out and check in documentation units and compounds in HOPE. The scripts do everything that the HOPE commands **checkin** and **checkout** do, plus the following:

- automatically compress and uncompress the files using the **gzip** and **gunzip** programs
- allow the use of wildcards, as in

```
doc_checkin *.doc -compound DOCproject-ug-src
```

The scripts take the same arguments as HOPE itself. For example:

```
doc_checkout -compound DOCsw-OEM-src -claim hard -recursive
-prompt
```

This checks out all the ScriptWorks OEM Guide source files and the subdirectory of graphics files and uncompresses them.

The following command compresses them and checks them back in:

```
doc_checkin -compound DOCsw-OEM-src -prompt
```

There is also a script, **doc_sync**, that is used to synchronize HOPE compounds for the doc group across all sites. This script is run automatically every night, so you should never need to run it yourself.

All scripts are checked out nightly into **~doc/share/src/** at all Harlequin sites where members of the documentation group are based.

3.2.1 Filename conventions

To make our files work with the **gzip** compression program, we have adopted (but not used from the beginning) the conventions listed in Table 3.2 for document file names. Note that these conventions also work within the 8.3 naming restrictions of some PCs. If you are working on a project for which 8.3 filenames are important for some reason, you can still use these conventions.

Table 3.2 Naming conventions for document files

File type	Uncompressed filename suffix	Compressed filename suffix
any	*none*	**`.gz`**
Frame document	**`.doc`**	**`.dgz`**
Frame book	**`.bk`**	**`.bgz`**
XWD (UNIX screenshot)	**`.xwd`**	**`.xgz`**
TIFF (PC or Mac screenshot or image)	**`.tif`**	**`.fgz`**
PostScript	**`.ps`**	**`.pgz`**
PCX (PC screenshot, less used)	**`.pcx`**	**`.cgz`**

If you check in a compressed file like **`man.dgz`**, when you check it out the script delivers the uncompressed **`man.doc`**.

To easily add new files using this convention, rename the uncompressed file with the appropriate single letter extension, for example **`man.d`**. Then the following command compresses it and gives it the appropriate 3 letter extension:

```
gzip -S gz man.d
```

Checking in **`man.dgz`** will now work properly; checking out using the script will deliver **`man.doc`** uncompressed to your directory. **`doc_checkin`** will then compress it and change that to **`man.dgz`** for checking in.

Note: You need not feel obliged to compress files before adding them to HOPE. The scripts work equally well with uncompressed files. Opinions differ as to whether compressing files actually saves disk space on HOPE servers, and since, in principle, disk space on HOPE servers is limitless anyway, compression should not be an issue that concerns us.

3.2.2 HOPE compound naming conventions

The documentation group stores its files in HOPE under the compound **DOC**. Subcompounds of **DOC** follow this style of naming:

```
DOCproject                          project name
  DOCproject-book                   book name
    DOCproject-book-pdf             Acrobat files
    DOCproject-book-ps              PS files
    DOCproject-book-html            HTML files
    DOCproject-book-src             Frame source files
      DOCproject-book-src-figures   image files
```

For example, the WebMaker documentation HOPE tree looks like this:

```
DOCwm                           WebMaker
  DOCwm-library                 library files
  DOCwm-ug                      User Guide
    DOCwm-ug-ps                 User Guide PS files
    DOCwm-ug-src                User Guide source files
  DOCwm-um                      CERN’s User Manual
    DOCwm-um-src                CERN’s User Manual source files
```

3.2.3 Details

The following is a fuller description of the **doc_checkin** and **doc_checkout** scripts:

In the HOPE compound **DOCshare-tools-src** you will find two Perl scripts, **doc_checkin** and **doc_checkout**. These scripts are checked out nightly into **~doc/share/src/** at the site you are based. These let you check files into and out of HOPE, respectively, automatically compressing and uncompressing files when necessary. You can use the scripts to check any files in or out, just as you would use the corresponding HOPE commands. (But if you know you are not checking in or out any compressed files, the scripts do take extra time.)

To use the scripts, either add **~doc/share/src/** to your path, or check them out of HOPE and put them into some directory that is already on your path, such as **~/bin**. If you check out your own copies, remember to make them executable: **chmod +x doc_checkin doc_checkout**. Of course you can name the files whatever you want.

Note: You must run the scripts on a UNIX system. They have been tested on a SPARCstation and an RS-6000, but they are not guaranteed to work everywhere.

The scripts contain mappings of compressed-file suffixes to uncompressed-file suffixes.

The **`doc_checkin`** command works as follows: If you check in an entire compound, the script collects unit names for that compound that end with any of the compressed-file suffixes. For each such unit, it looks in the compound's directory for a writable file that matches the unit name (for example, **`foo.doc.gz`**). If it finds one, it checks in that file. If it doesn't find one, it looks for a writable file with the same base name but with the corresponding uncompressed-file suffix, if any (for example, **`foo.doc`**). If it finds one, it puts that file on a list of files to be compressed before the check in.

If you explicitly check in one or more units, **`doc_checkin`** does the same thing with the given unit. However, before starting its search, it examines the unit name given on the command line. If that name matches an existing unit except that the name given on the command line ends in the corresponding uncompressed-file suffix, the command uses the actual unit name for the check in. For example, if the unit **`foo.doc.gz`** exists and the unit name on the command line is **`foo.doc`**, the command checks in **`foo.doc.gz`**. It still conducts its file search as described above (looking first for **`foo.doc.gz`** and then for **`foo.doc`**).

The checkout command works as follows: After the HOPE checkout, the command searches the checked-out directories for all newly modified files that end with any of the compressed-file suffixes. The command then uncompresses these files.

If you explicitly check out one or more units, the checkout command examines the unit name given on the command line and, if necessary, checks out the actual unit that has the corresponding compressed-file suffix. For example, if the unit name given on the command line is **`foo.doc`** but only unit **`foo.doc.gz`** exists, the command checks out **`foo.doc.gz`**.

Arguments are generally the same as those to the corresponding HOPE commands, and HOPE arguments are passed to the HOPE **`checkin`** or **`checkout`** command. The scripts use the same algorithm as HOPE to determine which directory to use. This means that in general you must be in the compound's directory or its parent directory.

For both scripts, there are two main differences between the script's arguments and the HOPE command's arguments:

1. Because of the script's heuristics for unit names, the user can give arguments derived from file names. Thus, you can use file globbing — for example, you can say **`doc_checkin -comp foo *.doc`** to check in all files in the current directory that end in **`.doc`**. You don't have to put **`-and`** between unit names. (This is a major feature.)

2. The script has a **`-prompt`** option. If you give this, the script prompts before uncompressing or compressing any files and before overwriting a read-only file when compressing or uncompressing.

   Note: In the absence of **`-prompt`**, the script *DOES* overwrite a read-only file when compressing or uncompressing. It always prompts before overwriting a writable file.

You can use the **`-filename`** option; the scripts try to do something smart with this. It has the same meaning as in HOPE, but, when it applies to a unit, **`doc_checkin`** examines its suffix to figure out whether the file needs to be compressed first, in the same way it examines unit names in the absence of **`-filename`**. When it applies to an entire compound, it represents the directory to use for that compound. To check out a file into the current directory, no matter what the name of the directory, use **`-filename`** with an argument of **`.`** (period).

The only HOPE feature that does not work is checkout with the **`-ci-date`** option. Such files will be checked out successfully, but the **`doc_checkout`** script cannot figure out when these files need to be uncompressed; you will have to do that yourself.

The scripts allow some customization via environment variables. The only two that you might want to change are these:

`DEFAULT_COMPOUND`

Names a compound to use instead of the currently selected compound when no compound is given on the command line.

`ZIP_TYPES`	A colon-separated list (like **`$PATH`**) of mappings from compressed-file suffixes to uncompressed-file suffixes. Each entry between colons can be either a compressed-file type alone or the mapping *zip-type*=*unzip-type*, where *zip-type* is a compressed-file suffix and *unzip-type* is an uncompressed-file suffix. White space around colons and **`=`** is OK. The default value is: **`.gz:.dgz=.doc:.bgz=.bk:.xgz=.xwd:.fgz=.tif:.pgz=.ps`**

You can use the **`DEFAULT_COMPOUND`** environment variable to construct a simple shell script for checking the Dylan book sources in or out. The checkin script would be as follows:

```
#! /bin/sh
DEFAULT_COMPOUND=DOCdylan-book-src
export DEFAULT_COMPOUND
doc_checkin "$@"
```

A Dylan-book checkout script would be the same, substituting **`doc_checkout`** for **`doc_checkin`**. Suppose you call this **`dyb_co`**. Then you could go to your Dylan book source directory and do **`dyb_co *.doc *.book`** to check out all the sources and uncompress them. Or you could do **`dyb_co -claim hard oo.doc`** to check out and claim the unit **`oo.doc.gz`** and end up with the uncompressed file **`oo.doc`**.

`doc_checkin` and **`doc_checkout`** have now been used extensively in the doc group, and seem to be pretty trouble free. If you should find a problem, please report it to the Documentation group.

3.2.4 Adding new documents to HOPE

The following are the basic steps for adding a new document or set of documents to HOPE.

We will use an example of a new book, a User Guide, to demonstrate the steps.

You can do HOPE commands from either the UNIX command line or from inside the HOPE client itself. For brevity, we will show the commands as you would type them inside the HOPE client. Note that if you issue commands from inside the HOPE client, as shown here, you cannot use the doc scripts. However, if you issue commands from the UNIX command line you can. (Thus, you could use **`doc_add`** *stuff* instead of **`hope add`** *stuff*.)

1. Make sure you have all the files you need in your working directory.
2. Start the HOPE client.

   ```
   > hope
   ```

3. Select the parent compound for your new compound. We will add this book to the **`DOCproject`** compound, which contains all the documentation for the project.

   ```
   select -c DOCproject
   ```

4. Create a new compound in HOPE for the User Guide.

   ```
   create -dir ug -c DOCproject-ug
   ```

   If the compound you are creating is to contain binary files by default, you should also specify the **`-binary`** keyword in the above command. The issue of binary files is discussed in greater detail at the end of this section.

5. Add the new compound to the parent compound, **`DOCproject`**.

   Specify the **`-reason`** argument to give the reason for adding this book to the project. Specify the **`-c`** argument for the full name of the parent compound, and the **`-subc`** argument for the full name of the new subcompound.

   ```
   add -reason "User Guide for project"
       -c DOCproject
       -subc DOCproject-ug
   ```

6. Create and add the **`ps`**, **`html`**, **`src`**, and **`figures`** subcompounds. Specify the **`-binary`** argument for the **`src`**, **`ps`**, and **`figures`** compounds.

   ```
   create -dir ps -c DOCproject-ug-ps -binary

   create -dir html -c DOCproject-ug-html

   create -dir src -c DOCproject-ug-src -binary

   create -dir figures -c DOCproject-ug-src-figures -binary
   ```

```
add -reason "PS files"
    -c DOCproject-ug
    -subc DOCproject-ug-ps

add -reason "HTML files"
    -c DOCproject-ug
    -subc DOCproject-ug-html

add -reason "Source files"
    -c DOCproject-ug
    -subc DOCproject-ug-src

add -reason "Image files"
    -c DOCproject-ug-src
    -subc DOCproject-ug-src-figures
```

7. Add all the files in the book to HOPE. Here you can use the **doc_add** script to glob the files together and add them all at once, instead of having to add each file individually. You need to exit HOPE and issue the script from the UNIX command line.

   Specify the **-c** argument for the compound the files belong in. You can specify the files you want with UNIX wildcards, or by naming individual files. List the files after all the other arguments.

```
> doc_add -binary
          -reason "User Guide source files"
          -c DOCproject-ug-src
          *.doc *.bk
```

8. If your manual already contains figures, add them to the appropriate compound. Again, you can use **doc_add** to do this.

```
> doc_add -binary
          -reason "User Guide screenshots"
          -c DOCproject-ug-src-figures
          *.tif
```

Of course, once you have HTML and PostScript versions of your manuals, you should add them to the appropriate compounds. See Section 3.2.5 for more information about adding HTML files to HOPE.

Some files, such as FrameMaker source files, pdf, ps, and gif files, are best entered into hope in binary format, whereas ASCII text and HTML files are best entered in text format. A compound has a default format, which is set either by specifying the **`-binary`** keyword when it is created, or by using the following command.

```
set -c DOCfoo-bar -binary
```

On the UNIX platform, whether a file is checked into Hope in binary format or not makes little difference in practice, but it can have an effect if the files are checked out on a Mac or PC platform.

The exact arguments to HOPE commands of course may change over time, but the above steps are current as of November 1996. See the *HOPE User's Guide* and *HOPE Reference Manual* for more information.

3.2.5 Dealing with HTML files in HOPE

There is a potential problem with keeping HTML versions of our documentation (or, more precisely, WebMakered versions) in HOPE, which is this:

> Because WebMaker creates filenames automatically, it is not possible to just check out old versions and check in new versions. Old versions have to be removed from the compound completely, and new units have to be added.

There is a wider issue here—whether we should take this approach at all—but this isn't the appropriate place to get into that debate. This is just a record of what you need to do, given that existing doc scripts aren't up to the task of removing large numbers of units from HOPE. There are more elegant ways of doing what is described here. You could probably remove the units in one step by using **`xargs`**, for example. This is left as an exercise to the reader;-)

Note: It would probably be a relatively easy matter to create a new doc script (based on **`doc_add`**) which removed units from HOPE. However, this is probably not a good idea. Removing units from HOPE compounds is an intrinsically dangerous process, and so it is not wise to make it too easy. The process described below, while not hard, is time-consuming and difficult to remember. You could not easily do it by mistake!

3.2.5.1 Removing old units from HOPE

This is the tricky bit. The steps are as follows:

1. Create a file, called **`units`**, that contains just the names of the units you want to remove. The following UNIX command will do this for you:

   ```
   hope status -c DOCfoo-bar -find-units ‘*’ -show units -format
   program | awk ‘$1=="unit" {print $2}’ > units
   ```

2. From the UNIX shell, run the following (assuming you run the bash shell):

   ```
   for unit in `cat units`
   do
   hope remove -c DOCfoo-bar -u $unit -reason "Type your reason
   here."
   done
   ```

Obviously, replace **`DOCfoo-bar`** with the correct compound name, and type a sensible reason. When creating the file of unit names, you can replace the **`*`** in the rune with any UNIX regular expression, should you just want to remove specific units (**`*.html`**, for instance). If you call the file anything other than **`units`**, you’ll want to modify the appropriate commands above, as well.

And lastly, you can run the commands in step 2 from a shell script if you want, of course.

For the average manual run through WebMaker, this will take a long time! Go and do something else.

3.2.5.2 Adding new units

This is much easier, although you still need to do it in two steps:

1. Add the HTML files (non-binary).
2. Add any GIFs, etc. (binary).

You can do this with the **`doc_add`** script, as follows:

To add HTML files:

1. **`cd`** to the directory containing all the files.
2. **`doc_add *.html -c DOCfoo-bar -reason "Type your reason here."`**

To add GIFs or other binaries:

1. **`cd`** to the directory containing all the files (although you're probably already in there if you're following directly on from the last step).
2. **`doc_add -binary *.gif -c DOCfoo-bar -reason "Type your reason here."`**

You may have to modify the phrase **`*.gif`**, depending on the sort of files you have, but it will usually be GIFs you're dealing with.

3.3 Using WebMaker

While not absolutely required, it is becoming more usual to run documents through WebMaker and place the HTML version on the doc group's internal Web site. Not only is HTML supplied to customers for an increasing number of our manuals, it also provides a convenient way of distributing review versions of documents internally. It is good practice, therefore, to create an HTML version of any document you release (either for internal or external consumption) and place the resultant files on our internal Web site, and in HOPE.

3.3.1 Creating the HTML

To create the HTML for any manual, you need to run the manual through WebMaker. You can do this on any platform (UNIX, PC, or Macintosh), although WebMaker is currently luded, which means that the latest UNIX release should automatically be available at any site; if you want to use the PC or Macintosh versions you will have to find out how to get hold of them yourself.

WebMaker can be found in the directory **`/usr/local/lib/webmaker`**. You need to make sure that the correct FrameMaker **`bin`** directory is on your UNIX path before starting the executable, which is **`/usr/local/lib/webmaker/webmaker`**. If you are using FrameMaker Version 5, this is /usr/local/frame/bin

The following Bash script should work. Save this script to a file, make the file executable and then run it either by typing its name at the UNIX command line or by **`source`**'ing the file.

```
#!/usr/local/bin/bash

echo “Running WebMaker on `hostname` ...”
echo “Checking contents of your PATH variable ...”

# Initialize some variables
OLD_IFS=$IFS
_matched=false
IFS=:

# Check PATH
for i in $PATH; do
    if [ “$i” = “/usr/local/frame/bin” ]; then
        _matched=true
        break
    fi
done

IFS=$OLD_IFS # Restore field separator

if [ $_matched = true ]; then
    echo “PATH looks fine.”
fi

# Amend PATH if necessary
if [ $_matched = false ]; then
    echo “Adding /usr/local/frame/bin to PATH.”
    PATH=/usr/local/frame/bin/:$PATH; export PATH
fi

/usr/local/lib/webmaker/webmaker
```

WebMaker, of course, comes with its own excellent documentation, complete with a very useful Quick Start chapter. However, the instructions given in this section specifically show you how to create an HTML version of any Harlequin manual with the minimum of fuss. These instructions assume that a FrameMaker book has been created for the manual you want to convert, and that the standard Harlequin templates have been used to create the files that are contained in that book.

To create an HTML version of any Harlequin manual:

1. Check the conditional text settings for your book. The settings must be correct for the version you are releasing.

In particular, make sure that WWW conditional text is visible, and that Paper conditional text is hidden. If you are creating HTML for use outside Harlequin, make sure that Comment conditional text is also hidden.

2. Save all the FrameMaker documents, including the FrameMaker book, in MIF format. Give each file the suffix **`.MIF`** (you must use upper case for the suffix).
3. Start WebMaker.

   Two windows appear: the main WebMaker window, and the Details window. You can largely ignore the Details window.
4. In the main WebMaker window, switch to the Includes view, by clicking on the View drop-down list box and choosing Includes.

   You use this view to include WML files for use when creating the HTML. A WML file contains a set of rules about how to convert each FrameMaker file to HTML, based on the names of the formats contained in the FrameMaker files. The doc group has its own WML file (**`hqn.wml`**) that is used to convert files that have been created using the standard Harlequin templates. This file is checked out nightly into **`~doc/share/template/wml`**.
5. Choose **Edit > Add Include** to add include a WML file.

   The WML Library dialog appears. This lets you choose the name of a WML file.
6. In the File text box, press **`Ctrl+U`** to delete the default text, and type **`~doc/share/template/wml/hqn.wml`**, followed by Return.

   The rules from the WML file are read into WebMaker. Next, you need to tell WebMaker the name of the FrameMaker book that you want to convert.
7. In the main WebMaker window, click **Make a Web**.
8. Check the following settings in the Make a Web dialog.

Frame Document	The name of the MIF file you want to convert. Specify the MIF version of the book in this field; WebMaker will automatically convert all the MIF versions of the files the book contains.
Destination Directory	The directory into which the HTML files are placed. It is often useful to place them in an **`html`** directory at the same level as the **`src`** directory in which your FrameMaker files are kept (to mirror their location in HOPE). If it does not already exist, create this directory at the UNIX command line (with **`mkdir`**), and then specify its name here.
File Name Prefix	A prefix for each file that WebMaker generates. By default, the prefix of the MIF file is used. Try to make this prefix both descriptive and short (six letters or less).
Main Web Title	A title that appears on every page that WebMaker generates. Specify the name and the version number of the manual. If the manual is a draft, specify this. If the manual is for internal use only, say so.
Filename Format	If you want WebMaker to generate 8.3 filenames, specify this here.

You can leave the Equations, Footnotes, and Debugging Markers setting as they are.

9. Click **Make a Web**.

WebMaker generates the necessary HTML and GIF files and places them in the destination directory. For large manuals, and manuals with many screenshots, this can take a long time.

When you quit WebMaker, you can discard the changes to the WML file unless you wish to save a version specific to the manual you are converting. This is unlikely to be necessary, unless you have had to make changes that modify the standard behavior of **`hqn.wml`** in any way.

3.3.2 Post-processing the HTML

Very often, the HTML that WebMaker produces is not as perfect as you would like. For example, you cannot specify a background color for your HTML files using WebMaker. In addition, pages produced by WebMaker occasionally run foul of bugs in WebMaker itself, producing HTML that is less than ideal.

Making manual changes to the HTML that WebMaker produces is time-consuming and undesirable; every time that a manual is regenerated, the same changes have to be made for the newly generated manual. The only effective way to make such changes is to provide a way to perform them automatically. The doc group has written a Perl script that lets you make all the changes you want to all the HTML files in a web in one operation. This script, called **`doc_html`**, lives in the same compound as the other doc scripts (**`DOCshare-tools-src`**) and gets checked out nightly into **`~doc/share/src`** at your site. If you have that directory on your UNIX path, then you should have access to **`doc_html`** automatically.

To use **`doc_html`**, just call it on the UNIX command line, as you would any of the other doc scripts. The syntax is as follows:

`doc_html [-help|-? -verbose -ix` ***file*** **`-bg`** ***color*** **`-hyp]`** ***filenames***

`-help`, **`-?`**	Show a usage message similar to this description.
`-verbose`	Print detailed progress information. This is especially useful if you are unsure of the changes you have specified, since you are given the opportunity to cancel the operation before any changes are made.
`-ix` *file*	Rename the index file to the standard name, and fix all links within the web. *file* is the name of the index file generated by WebMaker. See Section 3.3.2.1 for more details.

-bg *color*	Set the background of each page to the hex value of *color*. If *color* is omitted, use the hex value for white. See Section 3.3.2.2 for more details.
-hyp	Remove **Ctrl+E** characters due to suppress-hyphenation markers in Frame variables. See Section 3.3.2.3 for more details.
filenames	The set of HTML files you want to edit. This will usually be *.html, but you can specify a discreet set if you wish.

Note: doc_html edits any HTML file, not just those generated by WebMaker, but its effects on HTML files that were not created by WebMaker is undefined.

The **doc_html** script is constantly under development. If you need to make a change to WebMaker generated files on a regular basis, then **doc_html** should be able to cope with it. In addition, the script is far from perfect, and probably contains many bugs. Feel free to mail the doc group with any comments you have on **doc_html**.

3.3.2.1 Renaming the index file

The index file that WebMaker generates is the last file in the web, and can therefore have any name, depending on the number of files that WebMaker generates. Just as with any other page generated by WebMaker, the index file for this manual might be **style-156.html** on one occasion, and **style-164.html** on another.

This is a nuisance for most people, and a problem for the Lisp group in particular, who have to rewrite code on each generation of a manual, so that LispWorks can access the manual's index correctly. We have therefore agreed to ensure that the index file of every manual we generate has a predictable name, using the following algorithm:

- For non-8.3 filenames, copy the index file to the file *prefix***-index.html**, where *prefix* is the prefix given to all files in the web. Thus, the index file for this manual might be **style-index.html**.
- For 8.3 filenames, copy the index file to the file *pref***-ix.htm**, where *pref* is up to the first five letters of the prefix given to all files in the web.

> Thus, the index file for this manual might be **`style-ix.htm`**. If uppercase filenames are generated, then the filename for the index file should be uppercase as well.

Copying the index files in this way is a solution, but not a particularly elegant one. For the web for this manual, there would be two copies of the index file: **`style-index.html`** would be available for anyone who needed to predict the name of the file easily, and **`style-164.html`** (or whatever) would be the version that was actually linked to from all the other files in the web. A far more elegant solution is to rename the index file, and then fix all the links within the other files in the web so that point to the renamed index file. This is what the **`-ix`** option to **`doc_html`** does. For example, to rename the index file for this manual to style-index.html, you might specify

```
-ix style-164.html
```

Note: Currently, **`doc_html`** only provides a solution for non 8.3 filenames. If you have generated a web using 8.3 filenames, you must still copy the index file by hand. A future version of **`doc_html`** will fix this.

3.3.2.2 Specifying a background color

You can specify any standard UNIX color (as specified in **`/usr/lib/X11/rgb.txt`**) using the **`-bg`** option to **`doc_html`**, but unlike many of the UNIX definitions, *color* must be all lower case, and all one word. For example, **`DarkSlateGray`** and **`dark slate gray`** are both legal UNIX color names, but **`doc_html`** only accepts **`darkslategray`**.

If you specify a color that is not one of the standard UNIX colors, then the color name itself is substituted in the HTML files, rather than the hex color value; you had better be sure that it can be interpreted correctly by any browsers that will be used to view the files.

To allow you to overcome mistakes easily, you can run doc_html several times on the same web, specifying a different color name each time, and the appropriate substitution will be made. For example, suppose you accidently type

```
doc_html -bg whitw *.html
```

Suppose further that you're not using the **`-verbose`** option, so the change is made without further interaction from you, and because **`whitw`** is not a standard UNIX color, the string **`whitw`** is itself used in all the files, rather than a hex value. When you realize your mistake, you can just run **`doc_html`** again, as follows:

```
doc_html -bg white *.html
```

and the change is corrected.

Of course, white is such a popular color as a background that it is the default setting for the **`-bg`** option. So you should really have just typed

```
doc_html -bg *.html
```

in the first place, and avoided the confusion, but you get the idea.

3.3.2.3 Removing Ctrl+E characters

If you define your own variables in FrameMaker, and you use the suppress hyphenation marker (**`Shift+Meta+Hyphen`**, or **`Esc N S`**) in their definitions, then WebMaker will not convert them well. This can be troublesome if you tend to use variables for product names or the names of related manuals, and you don't want those names to be hyphenated in the paper or PDF versions of the manual.

WebMaker converts the suppress hyphenation symbol in a variable to a **`Ctrl+E`** character. In most web browsers, this appears as a square character. The **`-hyp`** option removes these characters from all the HTML files you specify.

Note: WebMaker deals with suppress hyphenation markers in the normal FrameMaker text flow correctly. You only need to use this option in the specific case where the markers are included in the definitions of any variables. You can also use this option safely on HTML files that do not contain the **`Ctrl+E`** character.

3.3.2.4 Some examples

The following call sets the page background to white, renames **`wibble-465.html`** to **`wibble-index.html`** and fixes the links in other pages, and removes any **`Ctrl+E`** characters. It works on all files in the current directory with the **`.html`** suffix.

```
doc_html -bg -ix wibble-465.html -hyp *.html
```

The following call sets the page background to peach, and removes any **`Ctrl+E`** characters. It works on all files in the current directory with the **`.HTM`** suffix.

```
doc_html -bg peach -hyp *.HTM
```

The following call sets the page background to white, and removes any **`Ctrl+E`** characters. It works on all files in the current directory that **`doc_html`** recognizes as being an HTML file. Currently, **`doc_html`** will only work on files whose suffix is either **`.html`**, **`.htm`**, **`.HTML`**, or **`.HTM`**. In particular, it will *not* corrupt any image files you have in the current directory.

```
doc_html -bg -hyp *.*
```

3.3.3 Publishing the files internally

Once the HTML has been created, and any post-processing you need has been completed, you should publish it on the internal Web site. That way, it can be browsed by any Harlequin employee in their browser of choice. Do this as follows:

For Lisp documentation:

1. Create a directory in **`~doc/public-html/lispworks/`** that will be a home to all the HTML files you want to publish.
2. Copy all the files created by WebMaker into the new directory.

   Typically, this will be a mixture of HTML and GIF files.
3. Edit the file **`~doc/public-html/lispworks.html`** and add a link to the first page of the manual, together with suitable descriptive text.

For ML documentation:

1. Create a directory in **`~doc/public-html/ml/`** that will be a home to all the HTML files you want to publish.
2. Copy all the files created by WebMaker into the new directory.

   Typically, this will be a mixture of HTML and GIF files.
3. Edit the file **`~doc/public-html/mlworks.html`** and add a link to the first page of the manual, together with suitable descriptive text.

For ScriptWorks documentation:

1. Create a directory in **`~doc/public-html/sw/`** that will be a home to all the HTML files you want to publish.
2. Copy all the files created by WebMaker into the new directory.

   Typically, this will be a mixture of HTML and GIF files.
3. Edit the file **`~doc/public-html/scriptworks.html`** and add a link to the first page of the manual, together with suitable descriptive text.

In order to edit the relevant HTML file in step 3, you should load it into your favorite text editor and edit the raw HTML. If you balk at the thought of doing this, don't worry. It is usually relatively simple to copy an existing link and change it to point to your files.

The contents of **`~doc/public-html`** are synchronized automatically overnight at all sites where there are writers.

3.3.4 Putting the HTML into HOPE

Whether you add the generated HTML to HOPE is a matter best decided on a case by base basis. If you are releasing a manual internally, for review purposes only, then you may decide it is not necessary. If you are generating HTML for sending to customers, then you probably will want to add the files to HOPE.

For some products, adding the files to HOPE may be a necessary part of ensuring that the files are included with the product, since distributions are often built by checking out HOPE compounds.

If you have created several versions of the same manual (for instance, one with comments visible, and one without), then it is safest to add the version without comments to HOPE.

There are a number of issues you need to consider when adding WebMaker generated files to HOPE. For details about these, see Section 3.2.5.

3.4 Converting a Frame book to an Acrobat file

This section describes how to create an Acrobat file from a FrameMaker book file. An Acrobat file, also known as a PDF file, is an electronic version of a book that can be read using the free Acrobat Reader program. The program is available for Windows, UNIX, and Macintosh.

The steps that follow give detailed instructions for creating a PDF file, but here is the general idea. First you open a book file in FrameMaker and use FrameMaker's **File > Print** command to save an Acrobat-friendly PostScript file. Then you process this new PostScript file with a distiller program to convert the PostScript file to a PDF file.

3.4.1 Creating PostScript for PDF

The first set of steps cover creating the PostScript file that serves as a source for the PDF.

1. In FrameMaker, open the book file for the book you want to convert.
2. With the book file as your active window, choose **File > Print**.

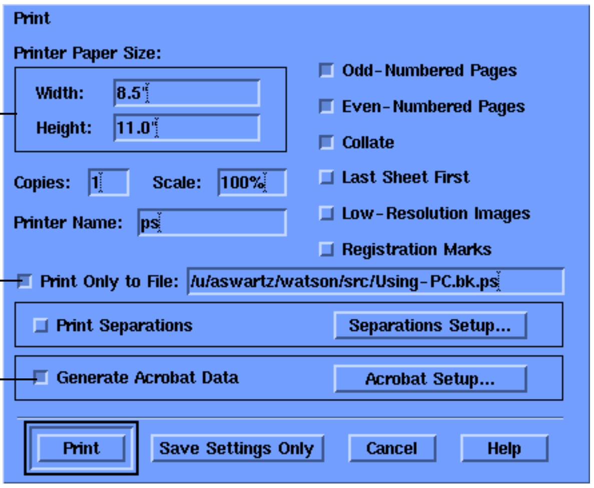
   A dialog box opens to ask you which files you want to print. Normally you will want them all.
3. Choose the files you want in the dialog box and click the **Print** button. Another dialog box opens showing many print options.

4. In the Print dialog box, set the options shown in Figure 3.1. Note that creating good PDF files requires that you use US Letter (8.5 by 11) as the printer paper size.

Set Width and Height to 8.5 by 11.

Turn on this option and note the file name in the text box.

Turn on this option as well.



Print

Printer Paper Size:
Width: 8.5
Height: 11.0

Copies: 1
Scale: 100%
Printer Name: ps

Odd-Numbered Pages
Even-Numbered Pages
Collate
Last Sheet First
Low-Resolution Images
Registration Marks

Print Only to File: /u/aswartz/watson/src/Using-PC.bk.ps

Print Separations
Generate Acrobat Data

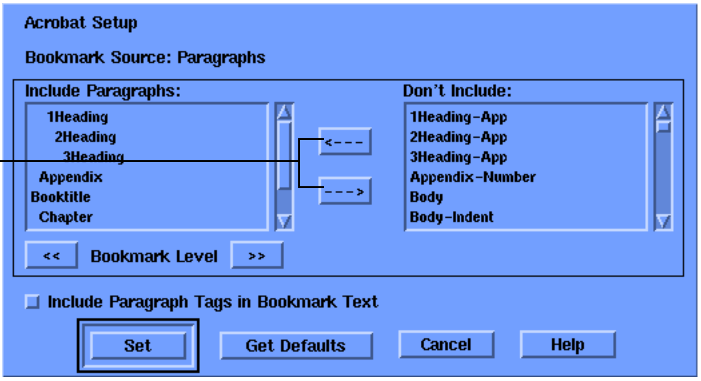
Figure 3.1 Print dialog box

5. Now click the **Acrobat Setup** button to open another dialog box where you can indicate which FrameMaker tags you want to use as different levels of Acrobat bookmarks, as described in the next step.

6. First, move items between the Include Paragraphs list and the Don’t Include list so that only items from the following list appear in the Include Paragraphs list. (To move an item, select it then click the arrow button that indicates the direction you want it to move, as shown in the picture below. To move all items at once, Shift-click the appropriate arrow button.)

```
1Heading
2Heading
3Heading
1Heading-App
2Heading-App
3Heading-App
Appendix
Booktitle
Chapter
Glossary
XIndex
Preface
```

Click these buttons to move selected items between the two lists.



Acrobat Setup

Bookmark Source: Paragraphs

Include Paragraphs:

```
  1Heading
   2Heading
    3Heading
 Appendix
Booktitle
 Chapter
```

Don't Include:

```
1Heading-App
2Heading-App
3Heading-App
Appendix-Number
Body
Body-Indent
```

7. Indicate what level of PDF bookmark each FrameMaker tag should have.

Following the order in the table below, select each item in the Include Paragraphs list then click the **<<** or **>>** buttons as many times as necessary to indicate the level of bookmark you want it to have in the PDF file.

Table 3.3

Level	Tag
Level 1 (all the way left)	Booktitle
Level 2 (indented just one level)	Chapter, Preface, Appendix, Glossary, XIndex

Table 3.3

Level	Tag
Level 3	1Heading, 1Heading-App
Level 4	2Heading, 2Heading-App
Level 5	3Heading, 3Heading-App

8. Once you have indicated all the bookmark levels, click the **Set** button to return to the Print options dialog box.
9. Click **Print** to save the PostScript file.

   It can take a few minutes to print a file. You know that the PostScript file is ready when it appears on your disk. It will be stored with the filename indicated in the print options dialog box, as shown on page 73.

3.4.2 Creating PDF

The second stage involves distilling the PostScript file into a PDF files. There are alternative procedures for A those using FrameMaker 5.5, B those using Distiller directly, and C those using Distiller through a watched folder. Follow one of these three procedures to produce the PDF file:

A Using FrameMaker 5.5

1. In FrameMaker Save the file as a PDF file rather than printing it as a PostScript file. You do not need to directly produce the PostScript at all. From the **File** menu choose **Save As**.
2. In the Save Document window, give the file a name, usually with the .pdf extension. Choose PDF in the Save as type box.
3. The Acrobat Setup that displays shows the bookmarks you set up in FrameMaker. Do not change them here. Keep your changes in the FrameMaker source. Click Set. Distiller will write a PDF file in the location you specified in the Save Document window.

B Using Distiller Directly

1. In Distiller choose **File > Open**.

2. Select the PostScript file to open and click **Open**.
3. Specify the PDF file name for output and click **Save**.
4. Distiller will display its progress until it creates the PDF file.

C Using Distiller Watched Folder

1. Next, leave FrameMaker and return to your UNIX shell or to Windows Explorer or whatever program you use to copy files.
2. Copy the postscript file you just created to the directory **`~doc/non-hope/acrobat/in`**.

   To copy the files from a Windows 95 machine, you must mount the directory by choosing **Start > Run** and typing

   ```
   explorer \\dorsal\doc\non-hope\acrobat\in
   ```

   In the US, replace the name **`dorsal`** with **`lantra`**.
3. If necessary, change the permissions for the file so the distiller can write to it.

   In UNIX, the command is

   ```
   chmod oug+w filename
   ```

   In Windows, you set the properties of the file by right-clicking its icon and deselecting the read-only option.
4. Wait 5 or 10 minutes. When the distiller is finished, it places the resulting files in the directory **`~doc/non-hope/acrobat/out`**.

   You will find your original PostScript file, the resulting **`.pdf`** file, and a **`.log`** file as well. Move these files wherever you like.

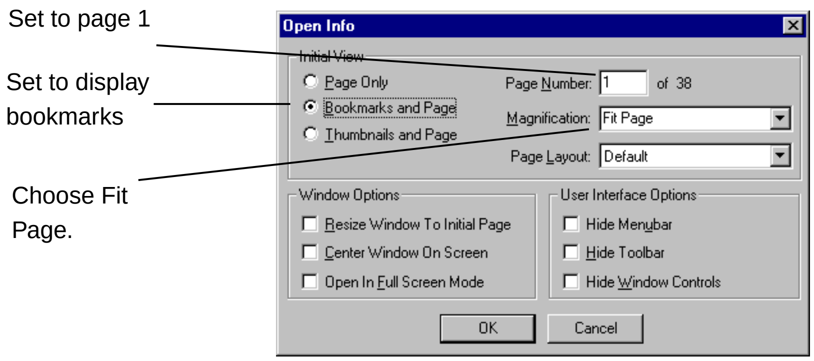
3.4.3 Postprocessing the PDF

To deliver PDF files to customers that are ready to use, we set the PDF file so when a user opens it, it displays page 1 with the bookmarks (table of contents) displayed down the side and sized to fit the window.

On UNIX systems running FrameMaker 5.1 this information on the initial state of the PDF display can be set in the PostScript file. In your **`fminit/`** directory install the PostScript file called **`ps_prolog`**. This writes this PDF information and makes several other useful changes to any PostScript file you generate. You can find **`ps_prolog`** in Hope **`DOCshare-template-postscript.`**

If you have not produced PostScript that uses **`ps_prolog`**, you will have to make these display changes to the PDF file you have created using Acrobat Exchange:

1. In Exchange open the PDF file.
2. In Exchange under **File** choose **Document Info > Open**.
3. In the Open Info dialog, set Initial View to Bookmarks and Page, set Page Number to 1, and set Magnification to Fit Page. Click **OK**.
4. Save the PDF file, now ready for distribution.



Initial View
( ) Page Only
(●) Bookmarks and Page
( ) Thumbnails and Page
Page Number: 1 of 38
Magnification: Fit Page
Page Layout: Default

Window Options
☐ Resize Window To Initial Page
☐ Center Window On Screen
☐ Open In Full Screen Mode

User Interface Options
☐ Hide Menubar
☐ Hide Toolbar
☐ Hide Window Controls

Figure 3.2 Exchange Open Info dialog

3.4.4 Pagination in PDF

PDF files have the same pages as the source PostScript files, and Acrobat Reader lets people navigate through the file by page. PDF files, however, are not aware of the page numbering scheme printed on the pages of your Post-Script file. PDF files are paginated from 1 through n.

This does not match the conventions of print publishing. The title page viewed in Acrobat Reader, for example, may be page 1 of 40. In the printed book the title page is implicitly page i and page 1 is the first page of text. For documents where this is likely to lead to confusion by readers who use the

Acrobat files as primary, Harlequin style will give up the print convention of restarting page numbering at the beginning of the text proper. These documents will display numbering in the front matter as lower case roman numerals, but the numbering sequence will be continuous from the first page of the document. The first page of Chapter 1 may be on page 9, for example.

3.4.5 Large File Bug Fix

A bug in Distiller causes some large PostScript files to fail in Distiller. The process runs but results in no PDF. To correct this, first process the file bigfile.ps in Distiller. It produces no output, but after running it Distiller can often correctly handle the large PostScript file. You can find **`bigfile.ps`** in Hope **`DOC-share-template-postscript.`**

4

Checklist for Finalizing Documents

When preparing a document to be printed for the customer, it is useful to follow a checklist to be sure that you clear up all the loose ends.

1. Turn off any conditional text that should not be seen by customers, and uncheck the Show Condition Indicators box. In any one document file, set the conditions as needed for the book (for hardcopy, show Paper only; for online, show WWW only). From the book file, choose **File > Import > Formats**. Be sure that the Import and Update area has *only* the Conditional Text Settings option checked. Update each file in the book.

   Warning: Use extreme caution in doing this, because you can easily destroy the formats for your document files by checking the wrong options in the dialog box.

2. Go through each file in the document with the Spelling Checker.

3. Use the Set Up File command in the book file on the TOC file to specify that the following paragraph types should be included in the TOC:

   Chapter, 1Heading, Preface, Glossary, Appendix, XIndex

   Note: If your document does not contain one or more of these, they will not be available as choices in the Set Up File command. That is OK; simply add the ones that are available.

4. Generate the book, and TOC and IX files.

5. As you generate the book, you might see messages about unresolved cross-references. Find them and resolve each one. Continue to generate the book until you see no such messages. (You can avoid a lot of this by making sure you keep the references updated, by doing **Edit > Update References** in your files as you are working on them.)
6. Check that the very first page is the title page.
7. Check that the very second page is the copyright page (and the appropriate one for your document). The copyright page has a number of fields that must coordinate with the title page of the document:

book title	
version number	This is the software version number rather than an edition number for the publication. This should be the major version number rather than minor release number, 4.0 rather than 4.0.2. Only the part number will tell the full story of the exact release this was written for.
date	The date of publication in the form Month Year, as in September 1997.
part number	Create or update the part number. See Section 1.12.
copyright date	Make sure it is or includes the year of publication.
trademark paragraph	This should include any Harlequin trademarks mentioned in the book. Adobe trademarks should also be listed. Other trademarks are covered by a general sentence, except in special cases.
DFAR paragraph	Begin with the name of the product, as in “The Watson Software...” or “The ScriptWorks Software....”

   Addresses and contact numbers should be verified.
8. Check that the very third page is the table of contents (TOC).

Note: There should be no extraneous blank pages between the title page, copyright page, and TOC.

9. Check that the index is listed in the TOC. If it is not, use Set Up File on the TOC from the book file to include the paragraph type XIndex.
10. Check that the preface, glossary, and appendixes (if any) are included in the TOC. If they are not, use Set Up File on the TOC from the book file to include the paragraph types Preface, Glossary, and Appendix respectively.
11. Check that the title page has no page number.
12. Check that the copyright page has no page number.
13. Check that the TOC's page numbers are Roman numerals, starting at iii.
14. Check that the Preface's page numbers are Roman numerals, continuing from the TOC.
15. Check that the first chapter's page numbers are numeric, starting at 1.
16. Check that the Glossary's page numbers are numeric (for example, 326).

    Note: The Preface and Glossary use the same template file, which by default is set up for a Preface (Roman numerals). If you use a Glossary, you have to modify the document format to get the numeric style. Details on the changes necessary to make a glossary out of the **`Preface.doc`** template are in the template file.
17. Check that the index has no missing pages (indicated by question marks in the entry where page numbers ought to be).
18. Do a visual inspection. Print the document or use Acrobat and examine every page of it. Check for typos, correct heading text, and so forth.

    Scan to be sure that the pages are all there, and are numbered correctly.

    Scan to be sure that figures have captions at the bottom, and tables have captions at the top.

    Scan that the prefix numbers of figure and captions are appropriately named.
19. Create a PS file of the book.

Depending on the nature of the release, you may want to create two PostScript files: One US Letter (8.5” x 11.0”) and one A4 (8.268” x 11.693”) size PostScript files.

In FrameMaker when you use Print to File to create PostScript, specify Generate Acrobat Data and setup the Acrobat bookmarks. See Section 3.4.

20. Create a PDF file. See Section 3.4.

21. Create HTML files.

22. HOPE step: Check sources into HOPE.

23. HOPE step: Check **`.ps`** versions into HOPE.

24. HOPE step: Check **`.pdf`** versions into HOPE.

25. HOPE step: If necessary, check HTML files into HOPE.

26. HOPE step: Do a checkpoint in HOPE of this version.

Appendix A

Formatting Lisp Code

This chapter is designed to provide helpful hints and guidelines for formatting Lisp code in documentation. Programmers have different coding styles, and there are many variations on basic rules and guidelines. This chapter presents the basic guidelines for writers, so that the code in your documents will conform to universally acceptable standards. This chapter includes general do's and don't's, many examples, and conventions for specific Lisp forms.

A.1 Problems and solutions

When confronted with code in a document you are working on, you can hope that the author of the code has correctly formatted it. You should avoid reformatting such correct code and code that exhibits only stylistic variations. But sometimes you will find true problems in the code. Some problems will be glaringly obvious — for example, everything is left justified — and some will be almost impossible to see — for example, the indentation is just a few spaces off. In the worst case, the code is completely misformatted, meaning you must start from scratch; perhaps you do not even recognize some of the expressions, yet you are still expected to produce a quality document. Fortunately, formatting is not as difficult as it might seem when you are looking at things that appear familiar to you. Let's begin by trying to make things less unfamiliar, with how to format simple Lisp forms. When formatting sections of code, always remember to use spaces, rather than tabs, to align the various elements. If you use spaces, the alignment is preserved when you run the document through WebMaker; this is currently not the case if you use tab characters.

The **cons** function is one of the simplest kinds of Lisp forms: a function call with two arguments.

cons ***object-1 object-2***

```
(cons 2 nil)
```

```
(cons 'a '(b c d))
```

When confronted with a piece of unfamiliar code, you can look for the function name and the argument list (or "lambda list") and, if it helps to improve the clarity, align the named arguments vertically.

```
(cons 'a-very-long-argument-name
      '(boringly complex details))
```

The same holds true for any subform of another form: indent its argument subforms, inside the containing form.

```
(cons a (list (car b) (+ b (car c))))  ; before

(cons a                                ; after
      (list (car b)
            (+ b
               (car c))))
```

If the argument list for a particular form is not clear, you can look in *Common Lisp: The Language* or the Common Lisp specification for detailed explanations and examples.

A.2 Functions

Function forms—as opposed to macro forms and special forms—can be formatted in two basic ways.

- If the series of argument subforms is textually short, the entire function call can fit onto one line.

```
(mapcar #'(lambda (x) (list (first x) x)) items)
```

```
(sort segments #'string-lessp :key #'third)
```

- If the function call's argument list is textually too long, the code should be broken up with each element of the argument list aligned with the previous one.

  `set-macro-character` ***char function non-terminatingp readtable***

  ```
  (set-macro-character #\[
                       #'read-scripture-reference
                       nil
                       *king-james-readtable*)
  ```

The code should also be broken up if the structure of the argument list is complicated and leaving it on one line makes it difficult to understand.

```
(+ (/ (* x (/ y 2)) 4) (* z (/ a 2) (- b (* 4 x))))    ; before

(+ (/ (* x                                             ; after
         (/ y 2))
      4)
   (* z
      (/ a 2)
      (- b
         (* 4 x))))
```

If certain arguments are both textually short (so that no visual confusion results) and closely related in a way that the adjacent arguments are not, they can be retained on the same line, with the rest on other lines. The arguments should still be aligned with the previous ones.

`set-dispatch-macro-character` ***disp-char sub-char function readtable***

```
(set-dispatch-macro-character #\# #\[     ; disp-char and sub-char
                              #'read-scripture-reference
                              *king-james-readtable*)
```

If horizontal space is really at a premium, you can move the arguments to a new line and align all of them farther to the left according to the same rule.

```
(set-dispatch-macro-character
  #\# #\[                                 ; disp-char and sub-char
  #'read-scripture-reference
  *king-james-readtable*)
```

In general, do not put unrelated arguments or arguments which are complicated expressions on the same line. In this example, the two arguments, although logically related, are too complicated to be clear when you leave them on the same line. You should separate them onto different lines.

```
(set-dispatch-macro-character
  (compute-char (x y (+ 1 z))) (compute-char (* 5 x))   ; BAD
  #'read-scripture-reference
  *king-james-readtable*)
```

A.3 Lambda lists

A lambda list specifies a set of parameters and a protocol for receiving values for those parameters. Each element of a lambda list is either a parameter specifier or a lambda list keyword. All required parameters must occur before the first lambda list keyword.

Most lambda list keywords do not affect the enclosing form's formatting (**`&allow-other-keys`**, **`&aux`**, **`&environment`**, **`&whole`**). You should format these keywords and their specifiers like any other parameter specifier: left–align each keyword underneath the previous.

A few lambda list keywords do affect the way you format the enclosing form. These are the keywords **`&optional`**, **`&key`**, **`&rest`**, and **`&body`**.

A.3.1 &OPTIONAL

If **`&optional`** is present, the optional parameter specifiers are those following **`&optional`** up to the next lambda list keyword or the end of the list. Knowing if a function or macro you are formatting has an **`&optional`** keyword will help you notice if it has too few or too many arguments.

The function **`float`** is an example of a function with an **`&optional`** keyword.

`float` ***number*** **`&optional`** ***prototype***

```
(float 1)

(float 1 2.5)
```

A.3.2 &KEY

If **`&key`** is present, all specifiers up to the next lambda list keyword or the end of the list are keyword parameter specifiers. There must remain an even number of arguments after **&key**, which are considered as pairs. The first argument in each pair is the name and the second is the corresponding value. Keyword arguments are thus formatted according to the pairs, with all pairs on one line, a pair on each line, or each element of a pair on different lines.

The function **intersection** is an example of a function with an **&key** argument.

intersection ***list-1 list-2*** **&key key test test-not**

```
(intersection list1 list2 :test 'equal)

(intersection list3 list4 :key 5
                          :test-not 'oddp)
```

A.3.3 &BODY

The **&body** lambda list keyword occurs only in macro forms (**defmacro**, **macrolet**, **define-compiler-macro**, **define-setf-expander**). It is identical in function to the **&rest** lambda list keyword, except that it informs text editors and certain pretty-printing functions that the remainder of the form is to be treated as a body and indented accordingly.

Most defining macros have bodies. Those body forms are indented two spaces in from the beginning of the enclosing form.

```
(defun foo (x y)
  (+ x y))                              ; body
```

If you see the phrase **&rest body** or **&rest forms**, you can infer that the programmer means **&body forms**, and you should indent the subsequent forms according to the rules for bodies. Do not change the code; just treat it as it was really meant.

The macros **defun**, **do**, **do***, **let**, **let***, **unless**, **when**, **with-output-to-string**, and **unwind-protect** are examples of forms with bodies.

A.3.4 DEFUN

The macro **defun** is formatted by placing all arguments before the **&body forms** on the first line; the body begins with the third form and is indented two spaces on the next line.

```
(defun foo-function (x y z)             ;function name, arglist
  (take-some-action x)                  ;&body forms
  (take-another-action y z))
```

If the argument list is long or complicated, you can break up the **defun** with the name and first argument on the first line, the other arguments on the second and subsequent lines, and the body indented back to two spaces.

```
(defun my-assoc (key                    ; function name, arg1
                 table)                 ; more args
  (find-if #'(lambda (entry)            ; body
               (equal key (first entry)))
           table)
```

Other defining forms beginning with **def**, such as **defstruct** and **defvar**, are formatted according to the same rule, as are the macros **when** and **unless**.

A.3.4.1 DO and DO*

The macros **do** and **do*** have the same general format as **defun**: Their bodies begin with the third form, which is then indented two spaces.

```
(do ((good (read input-stream nil nil)      ;var/init
           (read input-stream nil nil))     ;step
     (test-forms *test-forms* (rest test-forms))) ;var/init/step
    ((or (not good) (not test-forms))       ;exit-test
     (if (or good test-forms)               ;exit-form
         ;; One ran out but not the other
         :data-mismatch
         :ok))
  (unless (equal good (first test-forms))   ;body
    (return :wrong-answer)))
```

A.3.4.2 LET and LET*

The special operators **let** and **let*** have a lambda list (“bound variable list”); their bodies begin at the second form, which is then indented two spaces.

```
(let* ((first (+ x y)) (second (+ first 2))) ; lambda list
  (...frob with first and second...))          ; body
```

If **declare** appears, it is indented as part of the body.

```
(let ((a 'inside) (b a))                      ; lambda list
  (declare (special a))                       ; declaration
  (format t "~S ~S" a b))                     ; body
```

When the arguments to **let** and **let*** get textually long, you break them up by aligning the various subforms. As always, indent the body (second form) by two spaces.

```
(let* ((diff (- newprice oldprice))
       (proportion (/ diff oldprice))
       (percentage (* proportion 100.0)))
  (list percentage 'percent 'change))
```

If you wish to emphasize the relationship between the values assigned to the variables, you can align them under each other (see Section A.6.1 for another example of this usage).

```
(let ((first-alphabet-half  '(a b c d e f g h i j k l m))
      (second-alphabet-half '(n o p q r s t u v w x y z)))
  (...do something to the alphabet...))
```

Some people always put the different clauses on individual lines, no matter how short or long the clauses are.

```
(let ((x 3)
      (y 4))
  (* x y (+ x y)))
```

A.3.4.3 UNLESS and WHEN

The macros **unless** and **when** have a condition test followed by the body.

```
(when t 'TRUE)

(when (eq a b)                                 ;test
      (print a)                                ;body forms
      (print b)
      (cons b eqlist))

(unless (member a alist)                       ;test
        (print a)                              ;body forms
        (cons a alist))
```

A.3.4.4 WITH-OUTPUT-TO-STRING and UNWIND-PROTECT

The macros **`with-output-to-string`** and **`unwind-protect`** have bodies beginning with the second form.

```
(with-output-to-stream (s fstr)
  (format s "here's some output")              ;body forms
  (input-stream-p s))

(let ((indent (get-indent stream)))
  (unwind-protect
      (progn (set-indent stream (+ indent 2))  ;protected form
             (do-some-output stream))
    (set-indent stream indent)))               ;cleanup form
```

A.3.5 &REST

If **`&rest`** is present, it must be followed by a single rest parameter specifier, which in turn must be followed by another lambda list keyword or the end of the list. **`&rest`** and **`&body`** cannot occur at the same level in the lambda list; that is, this usage is incorrect:

```
(defmacro foo (x y &rest args &body forms))  ; incorrect
```

But this usage is valid, since lambda lists can be destructured into multiple levels:

```
(defmacro foo ((x y &rest args) &body forms))  ; OK
```

`&rest` and its parameter specifier usually appear as the last elements of the lambda list. Sometimes, **`&rest`** and its parameter specifier will be followed by **`&key`** and a series of keyword parameter specifiers. As previously mentioned, these arguments must be even in number.

The function **`gcd`** is an example of a form with an **`&rest`** argument.

`gcd &rest` ***integers***

```
(gcd 1 2)

(gcd 3 102 34 1006 10)
```

A.4 Macros with tagbody forms

A macro with a tagbody allows you to use **go** tags. Some frequently used macros with tagbodies are **prog**, **prog***, and **tagbody**. The indentation for such macros is special: each tag is left–aligned with the next inside the function; the statements are usually aligned as well, with the result that long tags have their statements moved to the next line.

```
(prog (var1 var2 (var3 init-form-3) var4 ...)
 top  (do-some-stuff)
      (if (maybe) (go there))
 here (do-some-funny-stuff)
      (more-stuff)
      (if (something) (go yet-another-tag))
 there
      (still-more-stuff)
      (if (probably-not) (go top))
 yet-another-tag
      (yet-another-set-of-stuff))
```

Some people right–align all the tags in a tagbody, especially when the tags are numbers or the tag names are long. Either way is acceptable, as long as it is consistent within each **prog** form.

```
(prog (foo1 foo2)
    1 statement1
   23 statement2
 7642 statement3
  961 statement4)

(prog (foo1 foo2)
   tagone statement1
   tagtwo statement2
 tagthree statement3
  tagfour statement4)
```

The **do** macro variants **do**, **do***, **dotimes**, **dolist**, **do-external-symbols**, **do-all-symbols**, and **do-symbols** have implicit tagbody forms, but they are rarely used.

A.5 Functions enclosing top-level forms

Some text editors only recognize top-level forms when they are in the leftmost column ("column 0"). So you have to separate the top-level forms from their enclosing special forms and put them at the beginning of the line. The closing parenthesis of the enclosing form will appear on its own line and will have a comment at the end saying what form the parenthesis is closing. (In real code, it is common for the comment to be the name of the form spelled backwards. For our purposes, the form's name should be spelled correctly.)

```
#+LispWorks

(progn
(defun foo ...)    ; top-level form
(defun bar ...)    ; top-level form
);ngorp - we should use "progn" here

(eval-when (test)
(defun ...)        ; top-level form
(defun ...)        ; top-level form
);eval-when
```

progn and **eval-when** are the most common enclosing forms.

A.6 Special forms

Some Lisp forms are exceptions to those already mentioned or have a special format. These include **cond**, **format**, **if**, **loop**, and **multiple-value-bind**.

A.6.1 COND and CASE

The macros **cond** and **case** consist of a series of condition clauses. Format **cond** by aligning its condition clauses with each other. The clauses begin with the first form.

```
;;; all clauses on same line
(cond ((null x) nil)                      ; clause1
      ((oddp (first x)) t)                ; clause2
      (t (anyoddp (rest x))))             ; otherwise

;;; long arguments split onto new lines
(cond ((null x) nil)                      ; clause1
      ((symbolp (first x))                ; clause2's test
       (cons (first x)                    ; clause2's consequent
             (extract-symbols (rest x))))
      (t (extract-symbols (rest x)))      ; otherwise
```

```
;;; most consequents split onto new lines
(cond ((null x) nil)                     ; clause1
      ((symbolp (first x))               ; clause2's test
       (cons (first x)                   ; clause2's consequent
             (extract-symbols (rest x))))
      (t                                 ; otherwise
       (extract-symbols (rest x)))       ; wastes vertical space
```

```
;;; all consequents split onto new lines
(cond ((equal person (first *friends*))   ; clause1's test
       'we-just-met)                     ; clause1's consequent
      ((member person *friends*)         ; clause2's test
       'we-know-each-other)              ; clause2's consequent
      (t (push person *friends*)         ; otherwise
         'pleased-to-meet-you)))
```

If you want to emphasize the relationships between tests or values, align the elements you wish to emphasize.

```
(cond ((equal test 1)  "O")              ; line up all the
      ((equal test 10) "X")              ;  consequents
      (t               " "))
```

The macro **case** is formatted similarly, except the condition clauses begin with the second form.

```
(case test-variable
  ((0 1) 'some-response)
  (2 'other-response)
  ((a b c) (list x)))
```

```
(case test-variable                     ; align consequents to
  ((0 1)   'some-response)            ;  emphasize relationship
  (2       'other-response)
  ((a b c) (list x)))
```

A.6.2 FORMAT

When multiple lines are necessary, put the function **format**'s first two arguments on the first line and subsequent arguments on other lines, also breaking up the lines if the arguments are long or complicated. The first argument, a stream, is almost always **t**, **nil**, or a single variable, and it is logically attached to the second argument.

```
(format t "~&~S squared is ~S"           ; format string
        number (* number number))        ; format arguments
```

Sometimes the first argument to **format** is a form which computes an output stream. Sometimes the form can get textually long. In this case, you can align the format string underneath the first argument on a new line, like you usually do with the other arguments.

```
(format (compute-stream *standard-output* *non-standard-output*)
        "~&~S squared is ~S"
        number (* number number))
```

A.6.3 IF

The **if** special form is usually formatted all on one line or broken up with the test, consequent, and alternative aligned.

```
(if (evenp x)
    'even
    'odd)
```

Some text editors will treat the alternative clause as a body and negatively indent it by two spaces, but this format irritates some people, interferes with clarity, and is generally discouraged.

```
(if (evenp x)
    'even
  'odd)                              ; discouraged
```

A.6.4 LOOP

When formatting the **loop** macro, you should align keywords introducing top-level clauses and indent keywords introducing subordinate clauses by two spaces. If a conditional or stepping clause is simple and short, you do not have to indent its subclause, like the **when** and **with** in this example.

```
(loop for x = 1 to 3                ; top-level clause
      with a = 3 and b = 4          ; stepping clause
      when (= x 2) do (print x)     ; conditional clause
      do (print b))                 ; top-level clause
```

But for clarity, and to reassure the reader that the conditional or stepping clause was completed on the previous line, you usually indent the subclauses.

```
(loop for i in '(312 234 453 657 768)   ; top-level clause
      when (oddp i)                     ; conditional clause
        do (print i)                    ; first subclause
        and collect i                   ; second subclause
          into odd-numbers
        and do (terpri)                 ; third subclause
      else                              ; conditional clause
        collect i into even-numbers     ; subclause
      finally                           ; top-level clause
        (return (values odd-numbers even-numbers)))
```

A.6.5 MULTIPLE-VALUE-BIND

The macro **`multiple-value-bind`** is formatted with its variables on the first line, the form generating the multiple values indented four spaces on the second line, and the body on subsequent lines.

```
(multiple-value-bind (symbol status)    ; variables
    (find-symbol symbol-name package)   ; yields two values
  (when (eql status ':internal)         ; body
    (process symbol package)))
```

A.7 Long lists

When you have a list that is very long, you can break it up in one of two ways.

- The first style is preferred for undifferentiated lists. Align each line of text with the previous line, inside the parenthesis.

```
'(this is a long string of text which will be returned as a
  value when the function is called with arguments of the
  wrong type or is called with too many arguments.)
```

- The second style is preferred when the initial token is a tag or label. Indent subsequent lines under the second element of the first line.

```
'(prepositions beneath in of on with from for to at same as like
               like about by beside around under above through
               amidst onto via inside outside behind over)
```

A.8 Plist style

Property lists—alternating sequences of names and values—can appear all on one line if they are short and simple enough. The macro **setf** and the special form **setq** use the Plist format.

```
(setq tag1 (intern (string-upcase tag1) *keyword-package*))

(setf (virtual-page-numbered 0) *hyperlink-top-page*)
```

You can represent short and alternating forms by either writing them all on one line if they are short enough, or beginning lines with individual pairs.

```
(setq level 0 state nil value 10)

(setq level 0
      state nil
      value 10)
```

If the name-value pairs are too long or complicated, you can choose to break them up in any of these ways.

```
;;; Start new lines with name-value pairs
(setq first-variable (mapcar #'some-function first-value)
      first-value    (list '(1 2 3) '(2 4 6)))

;;; Start new lines with name-value pairs
;;; Wrap complicated arguments onto new lines
(setq first-variable (mapcar #'some-function
                             first-value)
      first-value (list '(1 2 3)
                        '(2 4 6)))

;;; Put each name and each value on a new line, and optionally
;;;  indent the value lines. Indenting value lines looks silly
;;;  when the variable names are one letter in length.
(setq first-variable
      (mapcar #'some-function first-value)
      first-value
      (list '(1 2 3) '(2 4 6)))

;;; Put each name, value, and complicated arguments on new lines
(setq first-variable
      (mapcar #'some-function
              first-value)
      first-value
      (list '(1 2 3)
            '(2 4 6)))
```

```
(setq *book-list* '((bible god)
                    (cltl2 steele)
                    (transducer shires)
                    (clos keene))
      *theme-list* '((bible religion)
                     (cltl2 lisp)
                     (transducer cross-compilation)
                     (clos object-system)))

(setq *book-list*             ; if you're really tight on
      '((bible god)           ;  horizontal space; you won't
        (cltl2 steele)        ;  usually see this in real code.
        (transducer shires)
        (clos keene))
      *theme-list*
      '((bible religion)
        (cltl2 lisp)
        (transducer cross-compilation)
        (clos object-system)))
```

A.9 Documentation strings

Documentation strings are retrieved by the function **`documentation`**. They can contain any information the programmer wishes to place there. Usually they are used to describe how the form is to be used. They are indented at the same level as the body of the form.

```
(defun foo (x y)
  "Returns the + of the two arguments."
  (+ x y))
```

Documentation strings retain whatever amount of whitespace is in them. So if you have a very long doc string that must run over multiple lines, you should align the second and subsequent lines at the left margin. If you indent them, they will have extra whitespace in the string when they are retrieved.

Text editors generally treat any open parenthesis ("**(**") in column 0 as a top-level form. If a long documentation string that must be broken over multiple lines contains an open-parenthesis which falls in column 0, the developer must place a backslash ("\") in front of the parenthesis to keep the editor from confusing the string with a top-level form.

```
;;; This docstring will cause some text editors to behave oddly.
(defun my-plus (x y)
  "This function is like +, only I wrote it to take two arguments.
(It's kind of a strange little thing, but it works.)"
  (+ x y))

;;; This docstring will not cause trouble in most text editors.
(defun my-plus (x y)
  "This function is like +, only I wrote it to take two arguments.
\(It's kind of a strange little thing, but it works.)"
  (+ x y))
```

The backslash is a syntactic quoting character in Lisp. Adding a backslash has no effect on a regular character. But in order to insert a double quote or backslash in a string, you must put a backslash in front of it.

If you are using code examples developed by others, keep an eye out for documentation strings that contain parentheses, double quotes, or backslashes. For the sake of clarity, it is best to remove such characters from your examples, unless you specifically need to demonstrate the use of the backslash character itself!

A.10 Comments

The following remarks and examples are taken from the Common Lisp specification.

A.10.1 Single semicolon

Comments that begin with a single semicolon are all aligned to the same column at the right (sometimes called the "comment column"). The text of such a comment generally applies only to the line on which it appears. Occasionally two or three contain a single sentence together; this is sometimes indicated by indenting all but the first with an additional space (after the semicolon).

A.10.2 Double semicolon

Comments that begin with a double semicolon are all aligned to the same level of indentation as a form would be at that same position in the code. The text of such a comment usually describes the state of the program at the point where the comment occurs, the code which follows the comment, or both.

A.10.3 Triple semicolon

Comments that begin with a triple semicolon are all aligned to the left margin. Usually they are used prior to a definition or set of definitions, rather than within a definition.

A.10.4 Quadruple semicolon

Comments that begin with a quadruple semicolon are all aligned to the left margin, and generally contain only a short piece of text that serve as a title for the code which follows, and might be used in the header or footer of a program that prepares code for presentation as a hardcopy document.

A.10.5 Examples of comment strings

This first example describes comment usage in context.

```
;;;; The title is preceded by 4 semicolons

;;; Descriptive comments preceded by 3 semicolons
(defun fib (n)
  (check-type n integer)
  (cond ((< n 0)
         ;; Comments preceded by 2 semicolons are indented
         ;; to the position a form would be at that point
         (error "FIB got ~D as an argument." n))
        ((< n 2) n)             ; When preceded by 1 semicolons
        (t (+ (fib (- n 1))     ; Align to the comment column
             (fib (- n 2)))))) ; Comment only applies to this line
```

The second example lists the same code, but using more appropriate comments.

```
;;;; Math Utilities

;;; FIB computes the the Fibonacci function in the traditional
;;; recursive way.
(defun fib (n)
  (check-type n integer)
  ;; At this point we're sure we have an integer argument.
  ;; Now we can get down to some serious computation.
  (cond ((< n 0)
         ;; Hey, this is just supposed to be a simple example.
         ;; Did you really expect me to handle the general case?
         (error "FIB got ~D as an argument." n))
        ((< n 2) n)             ; fib[0]=0 and fib[1]=1
        ;; The cheap cases didn't work.
        ;; Nothing more to do but recurse.
        (t (+ (fib (- n 1))     ; The traditional formula
              (fib (- n 2)))))) ; is fib[n-1]+fib[n-2].
```

A.10.6 Additional remarks

In source code, a space between the final semicolon and the comment text itself is optional.

```
;;;This is a comment.
```

```
;;; This is a comment.
```

In documentation, however, you should strive to include the space, since it enhances clarity.

A.11 Last resort formats

You can format code very tightly in situations where horizontal space is really at a premium. But it is generally considered a “last resort”; avoid it whenever you can. It is better to make the code go vertical (that is, break up the lines of code onto new lines) when horizontal space is getting tight.

```
(foo
 (bar
  baz))
```

If all else fails, you can take the code into EMACS or the LispWorks editor and format it there. Bring the code into an empty buffer and do **`Meta+X lisp-mode`**. You can then go to each top-level form and do **`Ctrl+Meta+Q`** to indent it properly. If you need to break up long lines, hit linefeed (**`Ctrl+J`**) in front of the open parenthesis of the form you wish to place on the next line. This will indent the form (mostly) correctly. Then you can bring the code back into the FrameMaker document. If you still have questions, find a programmer who can tell you if it is formatted correctly. (It is worth noting that any technical document usually undergoes a technical review prior to publication, designed to catch and correct technical flaws in the document, so if you can't figure out how to make it right, make a note, and the technical reviewer should be able to help.)

A.12 Bad formats

Obviously, code should never be completely left–, right–, or center–justified. Likewise, it should not be all on one line unless it is a short, simple expression.

When you break up a form into pieces, you should not go back to having the form all on one line; you should not close one element on a line and include other elements on the same line. You should put subsequent list elements on separate lines. Not doing so not only confuses styles, but will confuse your readers. So you should not do this:

```
(eat '(apple
       coconut) donut)  ; discouraged
```

You should do this instead:

```
(eat '(apple
       coconut)
     donut)             ; OK: second arg aligned with first arg
```

A rule frequently broken says that anything that is inside a balanced set of parentheses must be indented at least to align with the opening parenthesis, and is usually indented one space more. It is never correct to indent a form less than the form that contains it.

```
(foo (bar
 baz))                  ; BAD
```

You should follow the appropriate rule for that form instead, or at least make sure the element aligns under the right parenthesis inside the containing form.

```
(foo (bar
      baz))            ; BETTER
```

A.13 Paragraph and character formats

Now that you’ve figured out what you’re up against, you need to know what paragraph and character formats to use for the code. In general, you can follow these guidelines, which apply to the Harlequin document format as of November 1996:

- Single lines of code take the Code-line paragraph tag.
- Two lines of code have the first line tagged Code-first and the second line tagged Code-last.
- Three or more lines of code have the first line tagged Code-first, the last line tagged Code-last, and all middle lines tagged Code-body.

When formatting code examples in reference manual material, you should use the RCode family of paragraphs. See Section 1.4 for more detail.

If you have a very long piece of code, and it needs to be broken up over pages, try to pick forms that obviously are top-level or begin something new. Typical such forms are **`case`**, **`cond`**, the **`do`** variants, **`let`** and **`let*`**, and **`progn`**. Do not break code in arbitrary places. If a place does not immediately present itself as reasonable, find a programmer who can tell you where it is acceptable to break the code.

Unspecified arguments in a line of code are tagged with the Variable character tag.

```
(+ var1 var2 &rest args)
```

Be sure that Smart Spaces and Smart Quotes are turned off in the FrameMaker document when you are formatting code. Single quote (“`'`”) and backquote (“`` ` ``”) mean different things in Lisp.

A.14 References

If you have questions about specific forms, you can find examples and explanations in these two books:

- *Common Lisp: The Language*, 2nd Edition, by Guy L. Steele, Jr.
- The Common Lisp specification

You can also start up a LispWorks and use the function **`function-lambda-list`** to see the argument list of a given function.

```
> (function-lambda-list ’if)
(TEST THEN &OPTIONAL ELSE)
```

Glossary

This chapter lists words with special Harlequin spelling or usage. It is divided into three sections: “Acronyms and trademarks” , “Harlequin spellings” , and “US spellings” .

Acronyms and trademarks

The acronyms and trademarks shown below are common in Harlequin documentation.

ASCII

All caps.

CLIM

Common Lisp Interface Manager. CLIM is a registered trademark of International LISP Associates (ILA).

CLOS

Common Lisp Object System.

DLL

Dynamic Link Library.

HCMS

Harlequin Color Management System.

HCS

Harlequin Chain Screening. Both the acronym and full name are Harlequin trademarks.

HDS

Harlequin Dispersed Screening. Both the acronym and full name are Harlequin trademarks.

HMS

Harlequin Micro Screening. Both the acronym and full name are Harlequin trademarks.

HPS

Harlequin Precision Screening. Both the acronym and full name are Harlequin trademarks.

HSL

Harlequin Screening Library. Both the acronym and full name are Harlequin trademarks. This is the package that includes HDS, HMS, and HCS.

HP-UX

HP has asked that we treat their trademark operating system name as "the HP-UX operating system" rather than as simply HP-UX and we have agreed in the Lucid contract. This does not require notice on the copyright page but determines our style in mentioning this product.

LispWorks

A Harlequin trademark.

MLWorks

A Harlequin trademark.

PC

Personal computer. Use PCs as the plural.

PostScript

An Adobe registered trademark. See Section 1.19.2.1 for Adobe's usage rules.

ScriptWorks

A Harlequin registered trademark.

UK

Rather than U.K., Great Britain, G.B., or GB.

UNIX

All caps.

US

Rather than U.S., U.S.A. or USA. Note that "U.S." is used in the context of the U.S. Government "Restricted Rights" notice; see Section 1.19.4.

Harlequin spellings

This section gives the spelling of certain words that are common in Harlequin documentation. For some words, several spelling variations might be possible; in such cases, the spelling used at Harlequin is shown. Other words are listed simply because you may need to look them up often. In many cases, the hyphenation of the word is important. You should also refer to Section 1.10 for more general guidelines on the hyphenation of common terms.

appendixes

Plural of appendix.

Boolean

Capitalized.

check box

Rather than checkbox or check-box. Treated other GUI "box" components similarly.

compile-time

When used in its adjectival form. There is no one word version in common usage.

database

One word.

dataset

One word.

double-click

Rather than “double click”.

drop-down

Rather than “drop down”. This refers to the term as used in “drop-down list box” or “drop-down combo box”.

e-mail

Use “electronic mail” only in contexts where e-mail might seem to be technical jargon.

indexes

Plural of index. Use indices only in the mathematical sense.

online

As opposed to “on line” or “on-line”. In its adjectival form, the one word variant of this is now sufficiently popular that we can consider using it. This is an exception to the guidelines given in Section 1.10.

on-screen

In its adjectival form, as opposed to “on screen” or “onscreen”. Needless to say, the two word variant should be used in examples such as “the dialog box appears on screen”.

pop-up

Rather that “popup” or “pop up”. As used in “pop-up menu”.

re-initialize

Rather than reinitialize, which looks odd to many people.

run-time

Although runtime is popular nowadays, it is important to to be consistent with the term compile-time, which cannot be a single word. Harlequin documentation often calls for discussion of the two terms in close proximity.

stand-alone

Rather than standalone.

submenu

Rather than “sub-menu” or “submenu”.

taskbar

This refers specifically to the WIndows 95 taskbar.

toolbar

Rather than “tool bar”. This refers specifically to a window that contains a number of buttons giving you quick access to different commands or tools. The single word variant is the one approved in the Microsoft guidelines.

tooltip

This refers specifically to the ballon-type help that often appears when you let the mouse pointer linger over a tool in a Windows application. It should really only be used in the context of Windows applications.

US spellings

Harlequin uses American spelling for most words. See Section 1.20 for a discussion of some of the more common differences between US and UK English. This section lists some of the more common US spellings.

among

Rather than amongst.

analog

Rather than analogue.

analyze

Rather than analyse. Also analyzing, analyzed.

artifact

Rather than artefact.

behavior

Rather than behaviour.

catalog

Rather than catalogue

center

Rather than centre.

check

Rather than cheque.

color

Rather than colour.

customize

Rather than customise. Also customizing, customization.

defense

Rather than defence.

dialog

Rather than dialogue.

ensure

To make certain, rather than insure.

generalize

Rather than generalise. Also generalizing, generalization.

gray

Rather than grey.

initialize

Rather than initialise. Also initializing, initialization.

judgment

Rather than judgement.

license

Rather than licence, as both noun and verb. Contrast with *practice*.

maneuver

Rather than manoeuvre.

meter

Rather than metre, as in meter, centimeter, millimeter.

offense

Rather than offence.

practice

Rather than practise, as both noun and verb. Contrast with *license*.

recognize

Rather than recognise.

specialize

Rather than specialise. Also specialized, specialization.

summarize

Rather than summarise.

tire

Rather than tyre.

while

Rather than whilst.

Index