The source for all things FrameMaker

Volume 1 · Issue 1

Welcome to *InFrame*

Welcome to the premiere issue of *InFrame*, the independent electronic magazine devoted to Adobe FrameMaker.

It is our hope to provide the FrameMaker community with a regular source of tips, information, and idea exchange.

Click here to read a few words from the people who brought you *InFrame Magazine*.

Features | Tips | Reviews | Case Studies | Adobe Speaks

Brought to you by Dennis Hays, Paul K. Schulte, and Larry Polk.

The source for all things FrameMaker

Features

Tips

Reviews

Case Studies

Adobe Speaks

A Personal Invitation from the Creators of *InFrame*

I have been a devoted user of FrameMaker since 1991 when I started a contract job for an aircraft maintenance and modification facility in Orlando. Florida. My first tasks were to scan pages from the McDonnell Douglas DC-10 maintenance manual and, after putting them through an OCR engine, save them as ASCII files. This, I would do each morning at 6:00am. By 8:30, there were three or four of us opening these files in FrameMaker on Macintosh computers and then spend then rest of the working day tagging the ASCII text. Click--Tag--Click--Tag--Click--Tag--Click--Tag. 100 pages a day tagging text and entering small changes for the aircraft we were modifying. I begged to do something different and my supervisor finally allowed me to modify the Frame templates into the revised ATA-100 specifications. I was hooked.

Since then, I've used FrameMaker on most every platform, designed templates for some of the largest organizations, and taught hundreds (or maybe thousands) how to use this application. And, for years now, I've diligently lurked on the FrameUsers.com list, coming out to speak my peace and then gently disappear again. Some of you may know me or remember the few tidbits I've offered over the years.

However, one prevailing item has crept into the discussions from time to time--the demise of *Frame of Reference*, the printed, quarterly newsletter that bound us Framers into a community. I, too, have wished for the return of *Frame of Reference*. And, I wondered why Adobe or someone else didn't start another, either by that name or another. A few weeks ago, I realized that I could be that someone, so I raised my "hand" in the list and said "I'll do it!"

This is a labor of love for me and for the two other volunteers putting this online magazine together. Larry Polk, the production designer has created a fantastic logo and designed the site; Paul K. Schulte, the production manager, has canvassed all of you requesting articles, tidbits of information, tutorials, and anything else worthy of inclusion into *InFrame*. The three of us receive no monies or any other compensation--we're just a team coordinating this effort to attempt to bring you a web-based periodical full of information you can use, whether you're just starting with FrameMaker or you are a Adobe Certified Expert (ACE).

The success of *InFrame* is dependent on two things, both based on your support. The first is that you read and use *InFrame* and the second is your contributions of material. If we don't have your contributions, we will not succeed and therefore you will be saying that there is no need for this type of communique. On the other hand, you can share your skills with every FrameMaker user in the world by letting us publish your findings, tricks, plug-in reviews, and anything else you feel is pertinent.

Dennis Hays, Publisher
(dhays@inframe-mag.com)
September 1999

And this whole adventure started out as a way to learn more and differently from the same old routine my department had been in for the first part of this decade. I started the Upper Midwest FrameMaker User Network (UM-FUN) in 1997 and still lead the UM-FUN into interesting areas of exploration.

I also wondered why *Frame of Reference*, or some such publication, hadn't re-appeared. Well, spurred on by the need for a replacement publication as we end this millennium, I helped start this ball a-rolling. And how far it rolls, we all shall see.

(I do seem to take on the projects that lead to more work in my quest for knowledge.)

I'd like to second Dennis' plea for your contributions. Whether we can succeed at making *InFrame* the success that the FrameMaker community yearns for, depends on these self-same contributions.

These contributions can run the gamut from a simple little technique that helps you do your work better or more easily all the way up to complex EDD's for FrameMaker+SGML. There is always someone out there who can benefit from sharing from your experiences. And even from your trials and tribulations!

This need for contributions is the same whether we're talking presenting at the upcoming FrameUsers Conference or to a local FUN,

or writing something that hopefully many of your colleagues will see. But taking the plunge can be very rewarding! You can feel satisfied that you've helped other users advance their training. And you can spur other more reticent users to contribute from their knowledge base.

So I expect all of you to keep me very busy this fall with many contributions to peruse.

Paul K. Schulte, Production Manager
(pkschulte@inframe-mag.com)
September 1999

I've been using FrameMaker for about 4 years now, and I learned the ins and outs of the program in a "trial-by-fire" fashion. I went to work for a software company that creates and markets engineering software. On my first day, the president of the company sat me down at a PC and basically said "...we use a product called FrameMaker to produce our documentation. It's very powerful. I don't know how to use it, nor does anyone else in the office. It's your job to rectify that situation." And so it began. At first I hated using it...it was quirky and nothing at all like my beloved Microsoft Word. But as we all know, adversity is the best teacher, and I rapidly developed a respect for the product. Later, after using Frame for a year or so, I began to actually appreciate it. Since that time, I've been hooked.

While I don't possess the breadth of experience with FrameMaker that my two colleagues have, I do match their dedication to the product. We've agreed that there is a great need for an informational outlet with respect to FrameMaker and this is what we are trying to give. I look back on my first experiences with Frame (I didn't even know there were any informational lists such as FrameUsers or Free Framers) and I can't help but think that my trials in learning this powerful and complex software would have been lessened, had there been a ready source of information exchange such as *InFrame*. I'm doing this because I believe in the product and I am devoted to promoting its usage and appreciation.

Larry Polk, Production Designer
(lpolk@inframe-mag.com)
September 1999

Features | Tips | Reviews | Case Studies | Adobe Speaks

Features

Features

Autonumbering in FrameMaker

by Dan Emory

Cross-Platform Shortcuts in FrameMaker 5.5

by Dave Valiulis

Tips

Conditional Text Overview

by Kay Ethier

Reviews

FrameScript: An Introduction to Writing Scripts - Part I

by Rick Quatro

Creating multiple autogenerating TOC's in bookfiles

by Tina Poole

Case Studies

FM+SGML Information Design

by Dan Emory

Adobe Speaks

Writing FrameMaker for Dummies

by Sarah O'Keefe

InFrame™

Features

Tips

Reviews

Case Studies

Adobe Speaks

Coming Soon: FrameMaker Tips and Tricks!

Features
Tips
Reviews
Case Studies
Adobe Speaks

Product Review: Visual Capture 1.0

Advanced Firmware Development

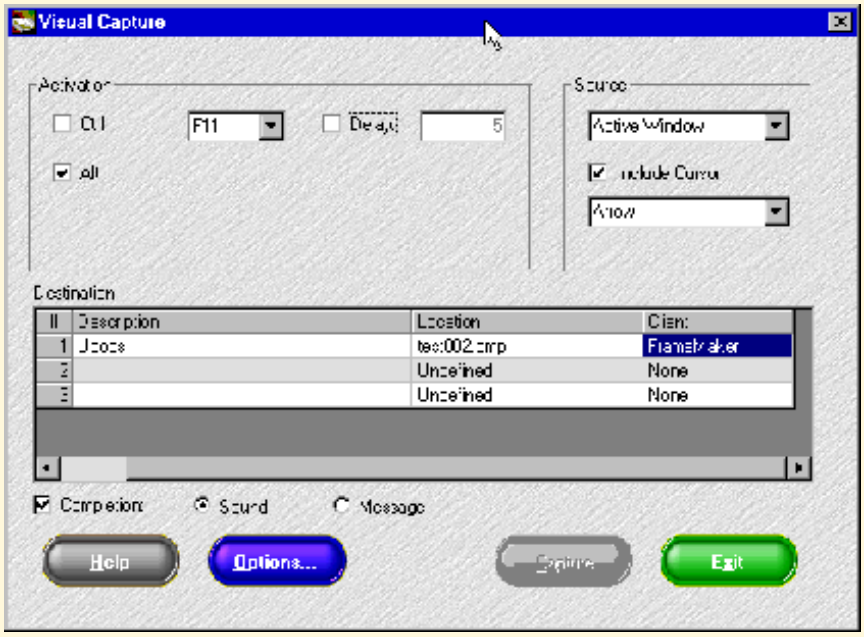
by Dennis Hays

Most of us using Adobe FrameMaker® have the opportunity, from time to time, to capture images from our monitor screen. And, if you've been a list member of either of the two FrameMaker® lists, or one of the other writing lists, you know that there has been considerable talk on which screen capture tool seems to work best.

For the most part, all of the tools work just about the same (with minor exceptions)--load the tool and load the program and/or files to capture, then capture the screens (I'm using the generic terminology "screens" here instead of "windows". Most all of these programs allow you to capture individual windows, parts of a monitor screen or any region, so in this review the word "screen" can mean any captured portion of what displays on a computer monitor.)

After a while, you've probably captured the screens you're going to document and placed these files in a folder. Some of the files, because of the nature of our business, need to be converted to other files types in an image editing program for use in web sites, black & white output, or color printing.

When done with the laborious process or converting image types, load FrameMaker® and, one-by-one, load the images into your application--usually by creating an anchored frame and then placing the image inside the frame and, horrors, re-size the image on the fly. I say horrors, because, as soon as you re-size an image, it usually doesn't conform to a multiple of the monitor dpi (96 dpi for Wintel monitors) any longer and you lose crispness.



Visual Capture - Main Window

I've found a tool that goes a long way to automating this process. Visual Capture 1.0, by Advanced Firmware Development (http://www.advfirmware.com) is a FrameMaker users "must-have" tool. Visual Capture allows you to capture screens and save them as multiple file types in the same session. So, with a single hot-key, you can create a color JPEG for the web, a grayscale bitmap for laser printing, and a color TIF file for color offset printing. One command and you've eliminated hours of work.

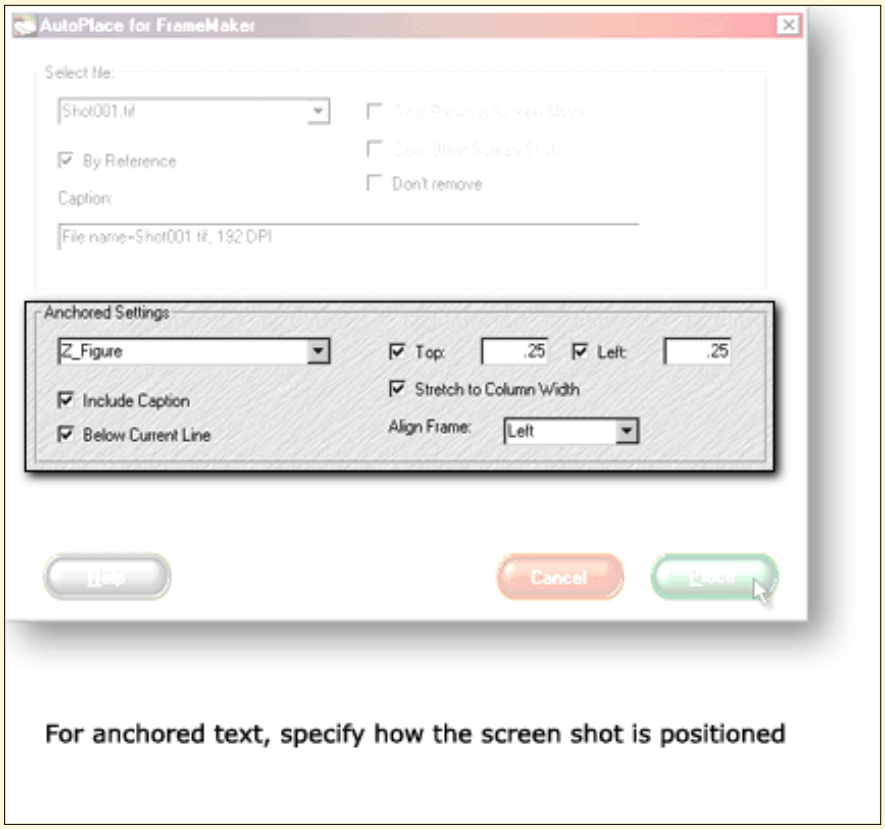
Quote from Advanced Firmware's web site: "Unfortunately, screen shots usually have to be 'cleaned up' with a graphics image editor like Photoshop®. Visual Capture takes care of graphics quality issues with features like optimization for 256 colors, automatic removal of moiré ("checkerboard") patterns, and assistance for correcting DPI resolution problems."

Visual Capture also:

- Reconfigures the Windows™ display system to remove moiré ("checkerboard") patterns in the Windows standard controls (such as in scroll bars and buttons--Visual Capture obviously cannot remove application-specific patterns).
- Assists you with calculating the proper DPI (dots per inch) based on an "even-multiple" of the display system's resolution
- Optimizes resampling for Indexed-8, 256 color results even if your display system is configured for true/high color settings (why set your high-resolution color display settings to accommodate low-resolution screen shots?)

If the above were the only differences between Visual Capture and the rest of the field, it would still be worth the price of admission (suggested selling price is US$99.95). However, if you're a FrameMaker® user (also PageMaker®, QuarkXPress™, or InDesign®), Visual Capture includes a plug-in that further automates the process. Visual Capture's plug-in automatically creates a new menu item (called GPS) on the FrameMaker® menu bar, allowing you to invoke the AutoPlace™ functions from within FrameMaker®.

Visual Capture's AutoPlace™ feature allows you save the link to a screen shot file into a workflow database. With a single menu command, you can place a screen shot, positioned and scaled as either an anchored or as an unanchored graphic, within the FrameMaker document.



- AutoPlace™ lets you select the next screen shot to be inserted
- Insert the image as an anchored graphic (below current line, inline with text flow) or unanchored (located anywhere on the page)
- For anchored graphic placement, the caption can be automatically inserted
- You control alignment and placement (depends on your DTP application's capabilities)
- Visual Capture keeps track of your last AutoPlace settings. So, the next time you select the Visual Capture > AutoPlace™ menu command, just click the Place button.

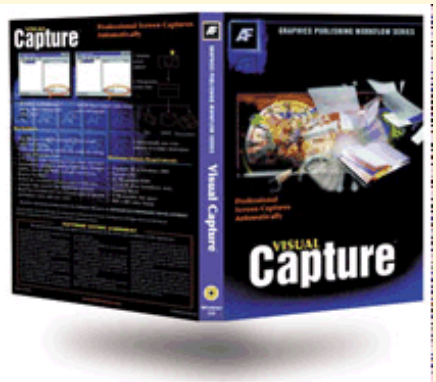
This is one of the first tools that has automated what has been a drudge procedure--capturing images from a computer monitor. I don't know why someone hasn't done something like this before, but, after looking and working with Visual Capture, that point is moot. It makes sense to be able to grab images from a monitor, in various flavors, with one command--especially since most of us are creating documentation for print, web, and on-line.

And, with FrameMaker's multiple task-oriented image placement--create an

anchored frame then import the image, it also makes sense to have a API (application program interface) that plugs into FrameMaker® to further automate the process.

Item	*Minimum*	*Recommended*
Operating System	Windows 98 (2nd Edition)	Windows NT (SP5)
Browser	Internet Explorer 4.01	Internet Explorer 5.00 (or later)
PDF Viewer	Acrobat 3.01	Acrobat 4.00 (or later)
CPU	Intel Pentium	Intel Pentium II
RAM	32 MB	64MB
CD-ROM drive (Install only)	2X	16X
Display Adapter	800x600 (256 colors)	1280x1024 (16-bit color)
Pointing Device	Mouse	Mouse or Tablet
Available Disk Space	32 MB	48 MB

Client	*Vendor*	*Versions*
FrameMaker®	Adobe	5.5.2, 5.5.3, 5.5.6
QuarkXPress™	Quark	4.03, 4.04, 4.10
PageMaker®	Adobe	6.5, 6.5 Plus (6.52)



Where to buy:

Advanced Firmware Development · **http://www.advfirmware.com**

PCConnection · (800) 800-5555 · **http://www.pcconnection.com**

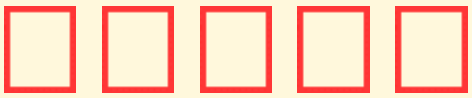
Publishers Toolbox · (800) 390-0461 · **http://www.pubtool.com**

DTP Direct · (800) 311-7084 · **http://www.dtpdirect.com**

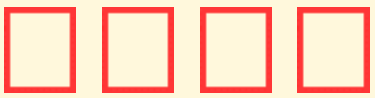
Review Score:

(4.5 Frames)

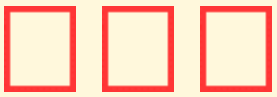
Legend:



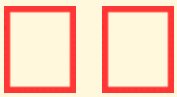
(Don't ask, just get it!)



(Add to your toolbox.)



(Worth a try.)



(Only if you really need it.)

(Not really worth the effort.)

Delphi Case Study: Motorola's FrameMaker Solution

by Nancy Mumford

Case Study: Creating Training Manuals Using Conditional Text - Part I

by Adam Korman

Features
Tips
Reviews
Case Studies
Adobe Speaks

FrameMaker's Next Release

Framers,

Back in Feb or March I told you that I'd keep you informed regarding the next release of FrameMaker as best I could and as soon as I could.

Wednesday Adobe presented its product roadmap during a press and analyst meeting held at Adobe's headquarters in San Jose. We didn't give any details other than the products and their schedule.

The next version of FrameMaker will be released in the 1st half of next year (2000). I know that's not alot of information, but for those of you planning your budgets for FY2000 this should help.

Also, this would be a good time for those of you who have let your FrameMaker maintenance lapse to get current.

Regards,

Mark

B. Mark Hilton
Group Manager, FrameMaker Marketing
Professional Publishing Solutions
Adobe Systems Incorporated
mhilton@adobe.com

This information was taken in its entirety from the FrameUser's mailing list (www.frameusers.com).

Features
Tips
Reviews
Case Studies
Adobe Speaks

Autonumbering in FrameMaker

by Dan Emory

This article is currently available in Adobe Acrobat PDF format. Click here to access the article.

Information: 6 pages, 0.226 MB.

To download the free Adobe Acrobat reader from Adobe's web site, click here.

Features
Tips
Reviews
Case Studies
Adobe Speaks

Cross-Platform Shortcuts in FrameMaker 5.5

by Dave Valiulis

This article is currently available in Adobe Acrobat PDF format. Click here to access the article.

Information: 30 pages, 0.609 MB.

To download the free Adobe Acrobat reader from Adobe's web site, click here.

Features
Tips
Reviews
Case Studies
Adobe Speaks

Conditional Text Overview

by Kay Ethier

FrameMaker's conditional text allows document creators to mark text, graphics, or table parts with special conditional tags. The tagged data may then be displayed ("show") or tucked out of sight ("hide") to provide the final versions. It allows authors to manage multiple documents from one file or book. Mixing "unconditional" text with the show- and hide-enabled text to combine documents saves time by maintaining more data in less documents, and can be utilized to jazz up electronic documents. Flexible and powerful, conditional text easily facilitates this repurposing.

Conditions applied to text saves time by maintaining more data in less documents. A document, or document set, may contain one or many conditions. The Adobe documentation does not note any limit to the number of conditions. (The maximum number in the author's experience is 50 conditions.) As with paragraph tags, the more conditions, the more management needed to avoid errors in "tagging." For users sharing files, it is important to give conditions names that are logical to all. Using condensed names or abbreviations are not recommended for shared files.

At first glance, new users tend to believe conditional text has one use: combining of similar documents. Consider these paper-publishing examples for "combining." Later we will examine repurposing examples and conditional text's flexibility.

Note: The below examples can be applied to more than one document type. One type is mention with each example.

- **Software documentation**
  Conditions adjust text and graphics for Windows, Macintosh, Unix, and Linux versions.
- **Corporate reports**
  Conditions are used to show engineers the technical details and show sales personnel revenue-related data without the technical minutiae.
- **Legal documents**
  Show or hide terms and conditions for each client--even each state.
- **Ditto for policies and procedures manuals**.
- **User guides**
  Multiple English versions (US/UK) maintained in one file or book.
- **Surveys**
  Conditional questions, subquestions, omits referrals ("skip to question 12") and saves paper by only showing appropriate questions.

Conditions are applied to text via the Special > Conditional Text dialog box. This dialog conveniently stays open on the desktop. Text selections allow application of single conditions, multiple conditions, or removal

("unconditional"). In addition to working through the conditional text dialog box, keyboard shortcuts allow certain actions (capitalization counts!):

- **CTRL-4**
  Select text, then use this to activate scrolling conditions list in the lower left corner of the document window. Scroll to select a condition and hit return to apply it to selected text.
- **ESC h C**
  With your insertion point within conditional text, use this shortcut to select the entire range of conditional text with same tag as your original insertion point.
- **CTRL-6**
  Select text and type this to remove all condition tags from the selection
- **CTRL-5**
  With text selected, use this to remove a single condition tag from text tagged with multiple condition tags. Type letter of tag to be removed and hit return

Once the conditions are applied to text, adjusting a single document is easy. Through the Conditional Text dialog box, hit the Show/Hide button. Select the condition(s) to show and those to hide.

Please note that at the book level, it is not necessary to change the Show/Hide for each file. Modify the Show/Hide properties for a single file, save it, and keep it open. Then, go to the book window and though File > Import > Formats import the Conditional Text Settings (the current Show/Hide Settings) through the entire book at once.

Note: If the files contain overrides, check only Conditional Text Settings in the Import Formats dialog box.

Once familiar with combining documents and tagging text, users may want to move into more involved examples--single files or books--based on the concept of repurposing. A savvy Frame user can use combining (above) and repurposing techniques without error.

- **paper + HTML**
  Wreate the paper version with conditional components that belong in the on-line version. These show-able Web components include navigation buttons, hypertext and index markers, even Web-optimized graphics (72 dpi instead of 150 dpi).
- **paper + PDF**
  While most projects in this category may be managed with dual templates, conditional text allows basic layout manipulation (including master page elements) that can push one template to do the work of two.
- **paper + HTML Help**
  Currently being tested, this document set involves blocks of conditional text hidden during the HTML production phase.

While conditional text is not difficult, its flexibility allows mistakes. A users first foray into conditional text should be a simple document. The most common mistakes are leaving too many spaces between words, or leaving extra returns outside the condition. Even advanced FrameUsers need to step lightly. With a little practice, though, the power of conditional text can expand the potential of many types of documents.

Features
Tips
Reviews
Case Studies
Adobe Speaks
InFrame™

FrameScript: An Introduction to Writing Scripts - Part I

by Rick Quatro

FrameScript is a lot like FrameMaker; it has a fairly steep learning curve, but once you learn it, you'll find it a real workhorse. The best way to start is to read the first three chapters of the *FrameScript Scriptwriter's Reference*. And the *Quick Reference* will give you a shortcut to learning the FrameScript syntax. This article will introduce some foundational concepts of writing scripts that will help you get started.

Before we start scripting, let's define a couple of terms. A FrameScript script manipulates FrameMaker *objects*. By objects, we mean items such as paragraphs, anchored frames, and pages. FrameMaker objects have *properties* that are characteristics of the objects. A paragraph has a *FirstIndent* property and a *Color* property. Many of an object's properties are the same as those accessed by the FrameMaker interface (although the names are often different); for example a paragraph's properties can be accessed through the Paragraph Designer.

Much of the work of scripting is finding out how to access an object and which of its properties to change. Let's see how this works by writing a short script.

Open a new, portrait document and type "Hello world!" in the first paragraph. Double-click the word Hello to highlight it. Choose FrameScript > Script Window to open the Edit Script Window. Type the following lines in the Script Window:

```
Set vText = TextSelection;
Display vText;
```

In the first line, we are setting up a variable called `vText`. In programming, a variable is like a placeholder for information. You use just about any name you want for a variable as long as it is not a FrameScript command or FrameMaker object or property name. I like to start my variables names with a lower case *v* to distiguish them from reserved names, and so that I know at a glance that they are variables. We are using the `Set` command and the equal sign to set the value of `vText` to `TextSelection`. `TextSelection` is a FrameMaker object that represents the current text selection in a document.

Click the Run button. The FrameScript command `Display` displays a message box showing you the value of `vText`. The contents of the message box may not seem very useful at this point, but it does tell use that we are displaying a `TextRange` object. Let's modify the script: change the first line to

```
Set vText = TextSelection.Begin;
```

and click Run. Now we are displaying a `TextLoc` (Text Location) object. A `TextRange` object is made up of two `TextLoc` objects; one marks the beginning of the selection (`Begin`), the other marks the end of the selection (`End`). Now

change the first line to

```
Set vText = TextSelection.Begin.Object;
```

and click Run. Now we are displaying a paragraph (`Pgf`) object. This is the paragraph that contains the text location (`TextLoc`) object which is part of the current text selection (`TextRange`) object.

This exercise illustrates another important concept: FrameMaker objects are usually nested inside other objects. FrameScript uses "dot-notation" to move up and down the list of objects. Once you locate the appropriate object, you can access its properties. Change the first line to

```
Set vText = TextSelection.Begin.Object.Properties;
```

and click Run. Now you see a list of the current paragraph's properties; a list that's probably too long to fit on your screen. Press Enter or Return key to dismiss the list. Change the word `Properties` to `Name` and click Run. The paragraph format that is applied to the paragraph is displayed.

Let's apply some of this FrameScript theory and write a useful script. This script will give you a sample of all of the paragraph formats in your document. It will insert one paragraph into your document for each of the paragraph formats in the document's catalog. The paragraph will contain the format's name and will have the format applied to it.

First, click New in the Script Window to clear it. Open a new, portrait document. The first thing we need to do is get a list of the document's paragraph formats. FrameScript has a special mechanism for accessing lists of FrameMaker objects. Type the following lines in the Script Window:

```
Loop ForEach(PgfFmt) In(ActiveDoc) LoopVar(vPgfFmt)
EndLoop
```

This script simply loops through the list of paragraph format objects (`PgfFmt`) in the document. At this point that's all it does; click Run and nothing happens. We want the script to do something to each object in the list. Let's use dot-notation to access each paragraph format's Name property. Add the following line to the script before the `EndLoop` line:

```
Display vPgfFmt.Name;
```

and click Run. Now the script displays the name of each of the paragraph formats. This can be a little tedious, because you have to click OK to dismiss each name. Change `Display` to `Write Console`, and click Run. Now the names are written all at once to the FrameMaker Console window. Maximize the Console window to see the list.

Writing a script usually means performing several tasks and stringing them together. We found out how to get a list of paragraph formats, but now our task is to get this list into the document. We will need a new paragraph for each of the paragraph formats in the document. To make a new paragraph, use the `New Pgf` command:

```
New Pgf NewVar(vPgf) PrevObject(vPrevObject);
```

This command requires a `PrevObject` (Previous Object) object; in other words, an object to place the new paragraph *after*. Our blank document only contains one paragraph, so we can make that our `PrevObject`. Delete the `Write Console` line, and add these lines to the script between the `Loop` and `EndLoop` lines:

```
Set vPrevObject =
ActiveDoc.MainFlowInDoc.LastTextFrameInFlow.LastPgf;
New Pgf NewVar(vPgf) PrevObject(vPrevObject);
```

and click Run. A new paragraph is added to the document for each of the paragraph formats in the document. The paragraphs are blank, so we need to add the paragraph format names. Add this line under the `New Pgf` line:

```
New Text Object(vPgf) vPgfFmt.Name;
```

The `New Text` command requires a location for the text; in this case, `vPgf` specifies the paragraph object that you just made with the `New Pgf` command. The text you are inserting is the `Name` property of the current `vPgfFmt` object. Delete the blank paragraphs in your document and try running the script. You should see a list of the paragraph formats in the document.

Before moving on to the next task--applying the correct paragraph format to each line--let's examine the script so far. The first and last lines simply loop through the document's list of paragraph formats. Whatever is inside the loop is repeated for each paragraph format in the list. This illustrates the role that variables play in scripting. The variable name `vPgfFmt` stays the same, but its *value* changes as the script executes. The same is true for the variable `vPrevObject`; each time through the loop it is set to the last paragraph (`LastPgf`) in the last text frame (`LastTextFrameInFlow`) of the main flow (`MainFlowInDoc`) in the current document (`ActiveDoc`). To help you visualize this, add the following line after the `Set vPrevObject` line:

```
Display vPrevObject.Text;
```

and click Run. You will see that the `PrevObject` variable is always set to be the last paragraph. Delete this line before continuing.

The third and final task is to apply each paragraph format's properties to the paragraph that bares its name. As you can guess, this task will be performed inside the loop. First, add the following line before the `EndLoop` line:

```
Set vPgf.Properties = vPgfFmt.Properties;
```

This line simply sets the properties of the new paragraph to match the paragraph format properties. Delete all the paragraphs from your document and run the script.

Before closing the Script Window, make sure you save your script with a name you can remember. I call mine `ShowAllParaFormats.fsl`. Next time, we'll introduce a different kind of loop and some more FrameScript commands.

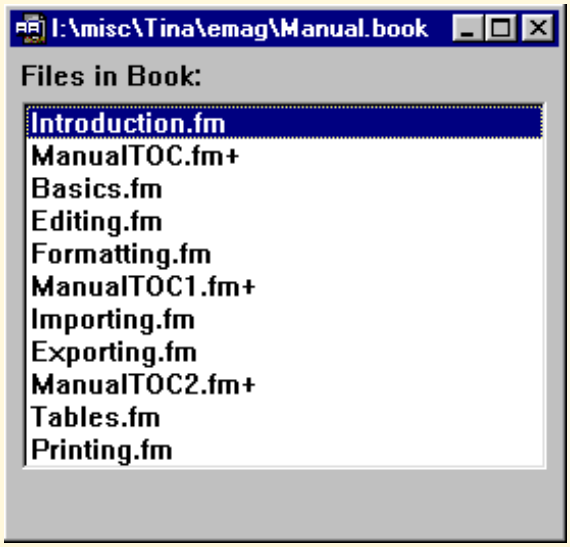
Features
Tips
Reviews
Case Studies
Adobe Speaks

Creating multiple autogenerating TOC's in bookfiles

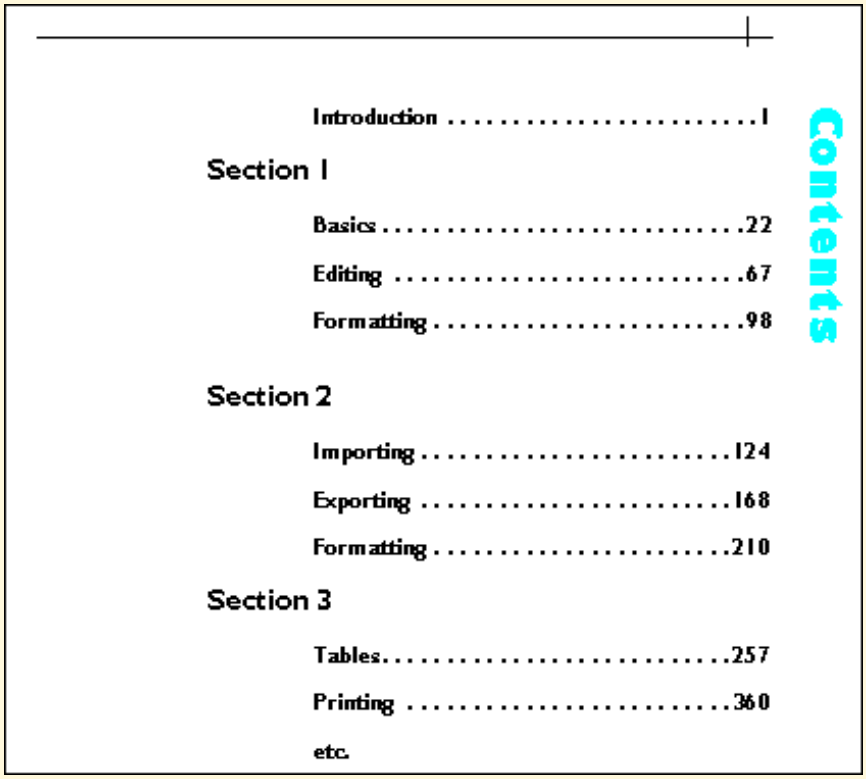
by Tina Poole

This method uses multiple TOC's that update automatically when the book is regenerated. Extra paragraph tags are needed but for the little bit of extra tagging required this is a big time saver once the document is setup

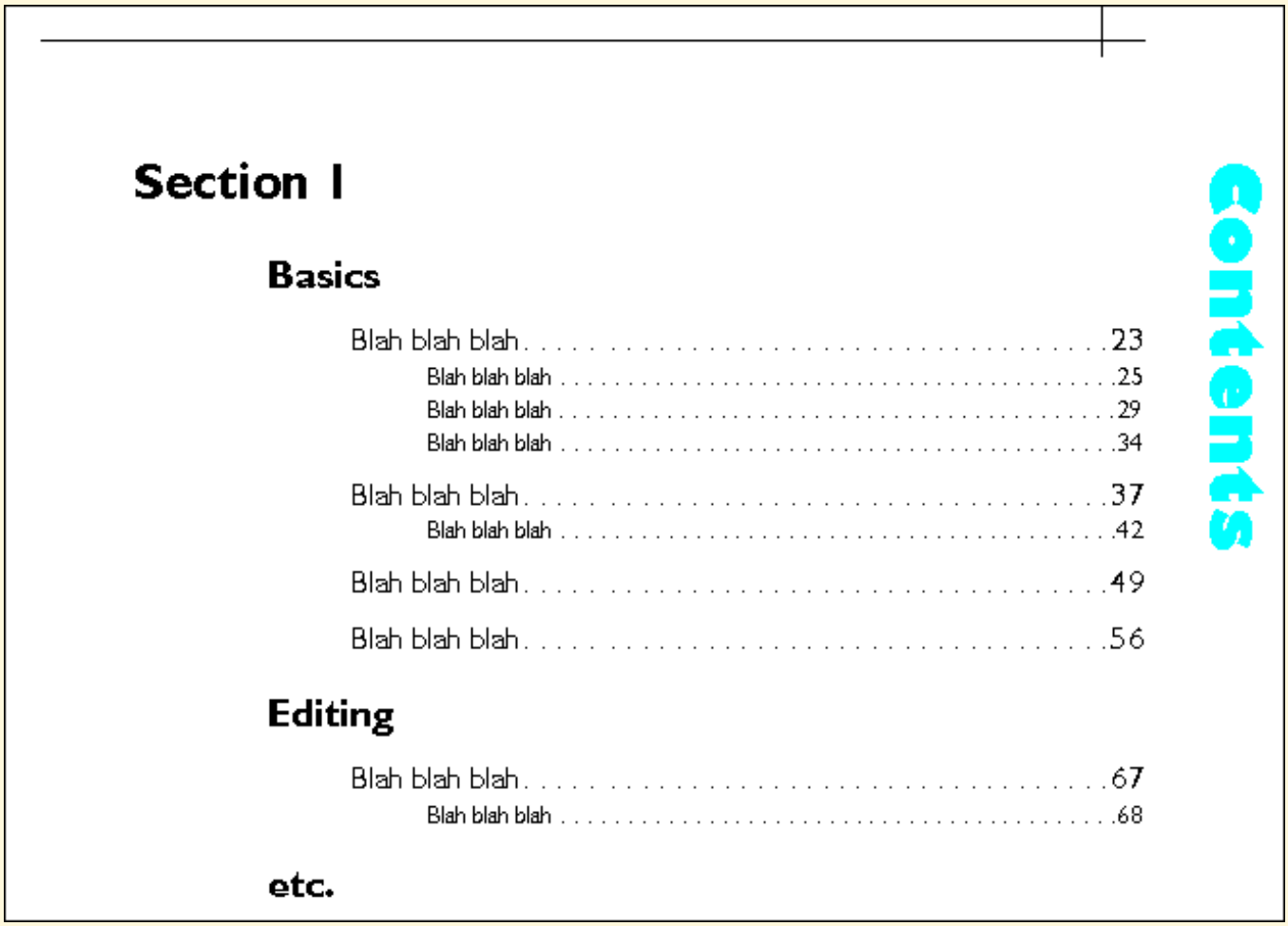
The book in this example has the following structure.



There is a master TOC



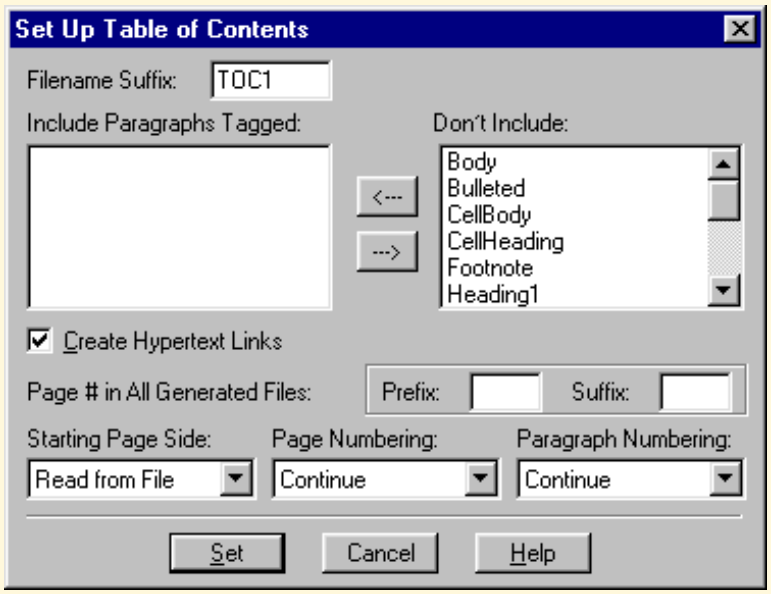
and each section has its own TOC.



First the files in each section will need to have unique paragraph tags, for this example the following is needed.

ChapterTitle (this tag is needed for the Introduction which is not in a section)	Section 1	Section 2	Section 3
	ChapterTitle1	ChapterTitle2	ChapterTitle3
	1Heading1	1Heading2	1Heading3
	2Heading1	2Heading2	2Heading3

Create the book file and add the main TOC. In the *Include Paragraph Tags* include all *Section* and *ChapterTitle* tags. Add another TOC but in the *Set Up Table of Contents -> Filename Suffix*: add a 1 after the TOC.



In the *Include Paragraph Tags* include Section1, Chapter1, 1Heading1 and 2Heading1. Add the other TOCs using the same procedure, i.e. TOC2 has all the *2 tags and TOC3 has all of the *3 tags. This will create the section TOCs. Generate the book as usual and all of the TOCs will be created. The section TOCs work exactly like any TOC and will update automatically when the book is regenerated.

Thanks to the Publishing Unit, Tests & Publishing, Queensland Board of Senior Secondary School Studies, Australia for this great time saver.

Features
Tips
Reviews
Case Studies
Adobe Speaks

FM+SGML Information Design

by Dan Emory

This article is currently available in Adobe Acrobat PDF format. Click here to access the article.

Information: 23 pages, 0.537 MB.

To download the free Adobe Acrobat reader from Adobe's web site, click here.

Features
Tips
Reviews
Case Studies
Adobe Speaks

Writing FrameMaker for Dummies

by Sarah O'Keefe

For eighteen months, I tried to convince a passel of publishers that a FrameMaker book was a good idea. I told them about the difficulty of learning FrameMaker; I eloquently described the dedication of a typical FrameMaker user; I emphasized the dearth of available books.

Apparently, this was not the right approach.

After about ten publishers, I decided that a FrameMaker book just wasn't going to happen. So I gave up and got busy with other projects. Six months later, out of the blue, IDG Books contacted me and asked me to write a Dummies book on FrameMaker.

Writing a Dummies book has pros and cons. On the good side, you can write just about as informally as you want, and bad puns are encouraged. One of the negatives is that you only have about 400 pages to work with—including the front matter and the index. Of course, it's impossible to cover the entire FrameMaker set in 400 pages (unless you use microscopic type).

Some judicious pruning was required. The Equation Editor was the first to go; it's an interesting and powerful feature, but used by very few users. Many Dummies books include an appendix with installation instructions, but not this one. I simply couldn't afford to give up 15—20 pages for it. I decided to focus on the core features that make FrameMaker so powerful. After some further investigation, I decided that this more or less corresponded to the items listed in the Import Formats dialog box, plus graphics.

I had a lot of trouble convincing the publisher that this approach made sense. Most of their editors have extensive experience with PageMaker and QuarkXPress, and so their instinct was to divide the book into sections on "how to do layout" and sections on "how to insert content." But eventually, I added some explanatory information to the beginning of the book and got to keep the FrameMaker-centric organization.

During the writing and editing process, I learned a couple of important lessons that I think can be useful to every FrameMaker user and advocate out there:

- Many people have never heard of FrameMaker, even in the publishing industry. They are much more familiar with the way PageMaker and QuarkXPress work. If you're working with someone who has desktop publishing experience, the FrameMaker approach is not going to make sense to them until you explain it.
- The publishing process is a lot like a hot dog factory. Even if you like the final result, you probably don't want to look too closely at how it's

produced.

After my experience writing this book, I'm more surprised than ever that computer trade book publishers don't use FrameMaker much. With the notable exception of O'Reilly & Associates, I do not know of a major publisher that uses FrameMaker to produce its books. And this caused me all sorts of grief. Here are some of the indignities that I endured:

- I had to produce text in—you guessed it—Microsoft Word.
- Graphics were shipped separately because they can't be embedded into the Word files. As a result, I couldn't see the graphics without switching to another application, which made the writing process more time-consuming.
- Figure and table numbers had to be typed in manually. By me.
- Cross-references to other chapters had to be created manually.
- Word's autonumbering for bullets and step lists was strictly verboten because it would not convert properly when the book went into production. Therefore, I had to type in bullets (asterisks) and step numbers, along with their tabs to line them up.
- No Word tables were allowed. All tabular information had to be set up as tab-delimited text so that it would convert properly.

If you ever get a bit miffed at FrameMaker (no pun intended), try working in Word for a few weeks. It'll take care of those traitorous impulses.

All in all, I'm pleased with how the book has turned out. But I do believe that I could have done it better, faster, and cheaper using FrameMaker. I hope that book publishers will take a hard look at their processes and think about the money they could save with better tools—and I can recommend a book for them to start with.

Features
Tips
Reviews
Case Studies
Adobe Speaks

Delphi Case Study: Motorola's FrameMaker Solution

by Nancy Mumford

40 Years of Space Heritage

Motorola's Satellite Communications Group (SATCOM) serves domestic and international government and commercial customers with integrated space communications systems and advanced technology applications.

From the earliest days of America's space program, even before Explorer I reached orbit in 1958, all U.S. spacecraft have maintained their vital link to Earth via Motorola SATCOM equipment. As the world watched Neil Armstrong take 'one giant leap for mankind,' they were doing so via a Motorola transceiver on the lunar module.

Problem: maintaining document accuracy with 1,000s of changing components.

Motorola's SATCOM team has hundreds of engineers, focused on designing, developing, and delivering new communication technologies. One of the big challenges this team faces is the management of product information as it moves from concept to design specification to product documentation. In each of these stages, the information is subject to review and change. Often, a change to the information means a change in the product itself.

To better manage completed documentation, Motorola uses an application called Compass, built with Open Text's LiveLink. This application provides information access to the entire team but fell short in managing "work-in-progress" documents. Motorola needed to find a solution that would allow them to reduce the approval cycle time, improve consistency, and better handle information reuse.

Information creation at Motorola is a very collaborative process. Each document is made up of text, diagrams, and images, all of which may be useful in other documents. To best optimize the creation, modification, and reuse of this information, Motorola determined that they needed a new approach that focused on individual components of information within each document.

The SATCOM team has to ensure the accuracy of 1,000s of highly sensitive information components, distributed across 10,000s of pages used in what is literally "rocket science." This problem exceeds the capabilities of traditional document management systems, which are designed to manage many-to-one relationships (e.g., many pages to one document). Rather it requires a system adept at handling many-to-many relationships (e.g., many components

comprising many different documents).

Needed: component-level reuse with automatic change notification

Given the impossibility of manually maintaining the accuracy of the many-to-many relationships between components and documents, an initiative was created to implement a "document creation system" enabling component-level reuse with automatic change notification, which would be integrated into the existing Compass system. A set of evaluation criteria was compiled for the document creation system which intentionally emphasized "creation" over "management" because of the intent to use it along with Compass. It was felt that the Compass system did a fine job of managing documents, yet clearly lacked the ability to create and assemble original documents from individual components.

This latter focus on reuse was the primary guide when evaluating potential solutions. Also considered was the authoring environment supported by the tools. It was decided that a tool was needed that could provide the functionality of SGML or XML, yet SATCOM's authors could not afford the steep learning curve imposed by editing directly within these environments. For this reason, the decision was made to standardize Adobe FrameMaker for the creation of documentation. Based on this criteria the search began for a tool with the following capabilities:

- provide a work-in-progress repository that would facilitate component reuse and allow parallel development of documents
- allow parametric data (dimensions, mass, tolerances, etc.) to be stored in an external database and accessible directly from the primary document editor, rather than hard-coded into documents
- eliminate uncontrolled and unmanaged reuse and manual "cut-and-paste" of changed components into child documents
- integrate directly with FrameMaker and provide conversion to XML

Solution: component management using Chrystal Software's Canterbury

After reviewing about a half dozen potential solutions that facilitated document creation, Canterbury from Chrystal Software was selected, based both on its sophisticated management of components and its support for FrameMaker. Using this system, SATCOM was able to streamline the document creation process by reusing components and facilitating collaborative authoring.

With the adoption of Canterbury, the SATCOM team is now able to route sub-parts -- individual sections, graphics, and diagrams -- of their information for approval based on automatic e-mail notification. In addition, they can reuse graphics and text without the burden of preplanning.

In the future, Motorola believes that tracking relationships between components of information will be critical. With Canterbury, they are able to determine where text and graphics are used throughout the documentation set. In addition, metadata (attributes) can be used to group related information.

Another efficiency came from reduced maintenance and infrastructure requirements as document components were referenced instead of replicated. Given the large file size of some components and the potential for their use in 100s or 1,000s of documents, the savings provided by referencing rather than replication is substantial.

Summary

SATCOM's document creation system illustrates the disconnect found with traditional document management, where the focus is largely (if not exclusively) on storage and retrieval, at the expense of the authoring process. Here, the SATCOM team found the greatest value came from not just managing "documents," but rather by allowing them to be decomposed into individual components and managed at the component level.

Features
Tips
Reviews
Case Studies
Adobe Speaks

Case Study: Creating Training Manuals Using Conditional Text - Part I

by Adam Korman

Introduction

The *FrameMaker User Guide* discusses two common uses for conditional text:

- Managing multiple versions of a book with minor differences (for example, a manual that describes software that runs on more than one platform)
- Inserting editorial comments into documents

This article describes how to use conditional text to achieve a more complex goal: to manage multiple versions of interrelated documents (in this case a set of training materials). The unique problem in this case (which served as the basis for this article) was that we wanted to include cross-references from one version of a book to multiple versions of the same book, but only maintain one set of files.

The article will cover a number of topics, including:

- How to organize the file structure for the documents
- How to create and manage multiple versions of interrelated documents
- How to create cross-references among multiple versions of the book while maintaining only one set of files
- Strategies and tips for addressing formatting issues
- Managing multiple versions and overcoming formatting issues by importing templates

Although many of the issues covered are unique to this scenario of interrelated versions of a conditional document, others apply to any kind of document. So, while some of the information presented here describes how certain obstacles were overcome in this particular case, much of the article focuses on the specific steps to take, and the various options available, when creating similar documents.

Prerequisite Knowledge

The basics of working with files in your operating system (deleting, copying and renaming files) are not covered in this article. Additionally, this article assumes that you are already comfortable with a number of FrameMaker's features, including:

- Working with text and other basic word processing features
- Creating and managing multi-file books
- Creating and managing generated lists (table of contents, index, etc.)
- Importing formats into documents and across a book
- Inserting and managing cross-references (including updating unresolved cross-references)
- Using and modifying paragraph tags
- Importing files into documents

This article will not cover the basics of using conditional text (creating conditions, applying conditions to text, showing/hiding conditions, etc.). Instead, it will describe how to use conditional text to create documents similar to those in this case. If you are not familiar with conditional text, you may want review the conditional text chapter in the *FrameMaker User Guide*.

Case Background

The client needed a training course for an audience which fell into two broad categories: entry-level staff and mid-level staff. Although most of the material applied to both audiences, we decided that it would be more effective to tailor the lectures and exercises to specifically address the experience level of each audience.

Basic Requirements

- Create two, similar versions of a training course ("version A" and "version B"); roughly 85% of the material would be exactly the same in both versions.
- Create a workbook ("WB") and leader's guide ("LG") for each version.

Problems and Considerations

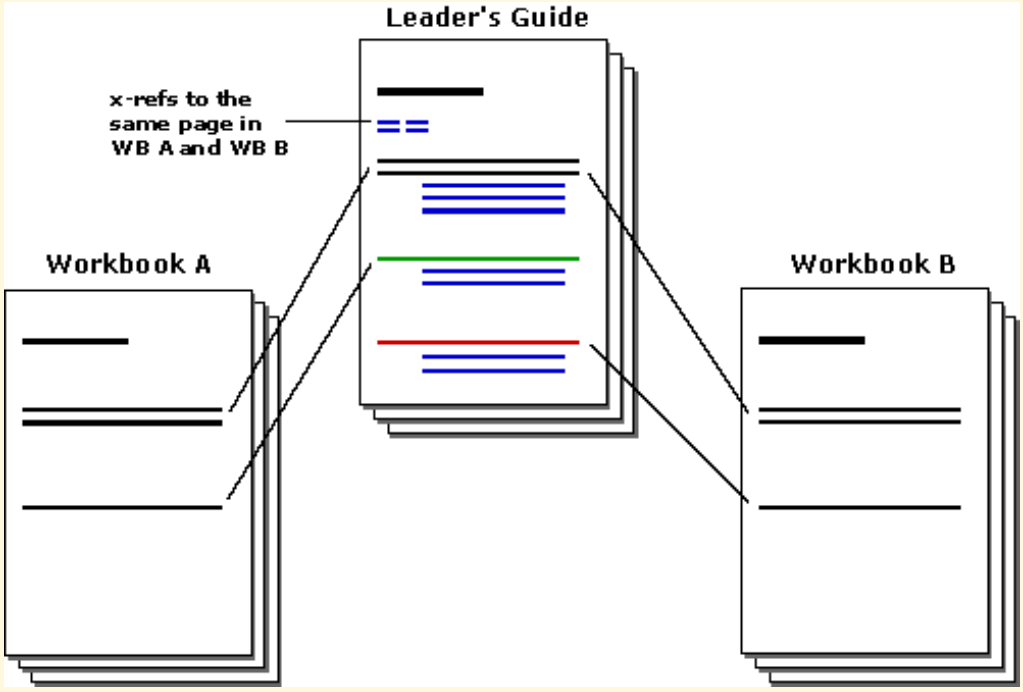
The most straightforward path to meet these requirements would be to create and maintain four books (a leader's guide and a workbook for each audience). This would have been problematic for a number of reasons.

1. **Maintaining the consistency of the material common to both versions**.
   In writing the material (and later in editing and updating the books), maintaining more than one set of files would require duplicate effort.
2. **Printing and other costs**.
   In this case, each trainer was responsible for teaching both versions of the course. This could mean providing up to four books for each trainer (both versions of the leader's guide and workbook). Additionally, the labor costs (in dollars and time) associated with writing and editing four books were an important consideration.
3. **Convenience**.
   It would be a hassle for each trainer to have to keep track of four books. Additionally, to keep a separate workbook at hand for reference while teaching would be physically unwieldy. More importantly, since the two versions would be so similar, for the trainer to make notations or comments in the leader's guide would have often required copying those notes into the second version.
4. **Potential confusion**.
   Since the two versions of the course were only subtly different in many instances, the trainers might not readily recognize these differences. Trainers could easily go into "autopilot" while teaching, and accidentally present the wrong version.

The Solution

The solution, which addressed each of these issues, was to create a single set of files from which all of the training materials were generated by using conditional text. This made it possible to provide the trainer with one book, rather than four, and still provide separate versions of the workbook to the two different audiences. Since so much of the material was exactly the same in both versions, it was actually preferable for the trainer to have a single book that clearly denoted the differences between the two versions.

- Since all the materials came from a single set of files, the workbook text was included in the leader's guide, obviating the need for the trainer to refer to a separate workbook while teaching the course.
- Instead of a "snapshot" of each workbook page in the leader's guide, we decided to intersperse guideline lecture scripts, comments and exercise instructions for the trainer in the appropriate context of the workbook text.
- Because the trainers periodically needed to refer participants to the correct spot in the workbook, and since leader's guide page numbering was different from the workbooks', we decided to include cross-references to the workbook pages throughout the leader's guide (a reference for every 1-2 pages of workbook text).



New Problems

Unfortunately, the "solution" introduced a whole new set of problems. Please be aware that while this article describes how to address these problems, the underlying issue (which does not go away) is that the whole procedure is complex. If you are not the only person involved, everyone who has to work with these documents should have a clear understanding of how to properly deal with the files and the potential pitfalls.

1. Initially, the most confounding aspect of the project was to find a way to maintain one set of files, yet include cross-references which would normally require three separate sets of files. The solution was to keep two sets of "dummy" files (one for each workbook) to serve as the source for cross-references, but which would not be edited in any way.

   Warning! Do not make any changes in the "dummy" files. All of your work will be lost.

2. Under normal circumstances, FrameMaker handles cross-references with simplicity and skill. In this case, you will become frustrated. Once cross-references are properly in place, they are completely stable and dependable, but reaching that point requires some extra effort.

   Warning! Make certain that everyone working on these files (including yourself!) fully understands the correct ways to insert cross-references. If not entered properly, cross-references will break in new and unpleasant ways.

3. Depending on your document layout, formatting conventions and templates, setting up conditional-text documents so that they do not require adjustment once you change the Show/Hide settings can be painstaking, difficult and/or frustrating.

   Warning! The hard work is worth the effort, and you should avoid taking the approach of making manual adjustments to the workbooks after you create (or update) them. If you have to make changes to the "dummy" files, your work will eventually be lost, and you will have defeated the

purpose of using conditional text.

4. Although one set of files is easier to maintain than four, many of the routine operations you take for granted with regular books will take more time.

   Warning! Allow extra time in your schedule. I highly recommend that well before your deadline you perform a trial run with a representative portion of all the books (at least two chapters) that includes proof-reading, editing, checking all cross-references, checking the table of contents and index, etc.

Getting Started

Most of the preparations and decisions required were the same for this project as for any other conditional text document. One of the primary concerns was deciding what conditions to use and how they would appear in the printed documents.

Setting up Conditions

In this case we required three conditions:

- LG for all instructions to the trainer
- WB A for text that only applied to version A of the course
- WB B for text that only applied to version B of the course

Any text common to both versions of the workbook was left unconditional (since it appeared in all three books).

Using Color

Although our budget did not allow for color printing, setting the conditions to display different colors made the process of writing and editing simpler. Before going to print we changed all conditions to appear black. But, even if your budget does allow for color printing, keep in mind the admonition that appears in the *FrameMaker User Guide* chapter on conditional text: if you use color to distinguish different versions of a document, you should also use some other visual cue, since many people have difficulty distinguishing colors (or are color blind altogether).

Additional Formatting

In this case, the trainers needed to quickly distinguish several kinds of information in the leader's guide:

- Instructions for the trainer (which did not appear in the workbooks) that applied to both audiences
- Instructions to the trainer that only applied to version A of the course
- Instructions to the trainer that only applied to version B of the course
- Text that appeared in both versions of the workbook
- Text that only appeared in version A of the workbook
- Text that only appeared in version B of the workbook

To convey all this information we adopted the following conventions:

- Since the workbook text common to both versions served as the "skeleton" of the training course, this text (and headers and footers) appeared in normal, black text.
- All LG text appeared in a font visually distinct from the one used for WB text. (If color printing had been an option, we would have also used a different color for all LG text.)

- WB text and related LG instructions that only applied to one version of the course were set off using special tables with headings that read either VERSION A ONLY or VERSION B ONLY. (Again, if color printing had been an option, we would have also used different colors for text specific to each version of the workbook.)

How to achieve this formatting will be discussed in depth in the next issue of *InFrame*, in the section titled *Formatting and Importing Templates*.

Writing and Editing

The most important thing to remember in this whole process is that all of the writing, editing and formatting was done in the leader's guide files. As you will see, any work performed directly in the workbook files would be lost.

File Structure

Because of the unique cross-reference problems in this case, you need to carefully plan how you will name and organize your files.

Naming Files

Although just about any naming convention you choose should work, using a simple identifier for each version of the book will make managing the files simpler. In this case, we chose to attach a prefix to each file name: "LG" for the leader's guide, "WBa" for version A of the workbook and "WBb" for version B of the workbook. So, files were named as follows:

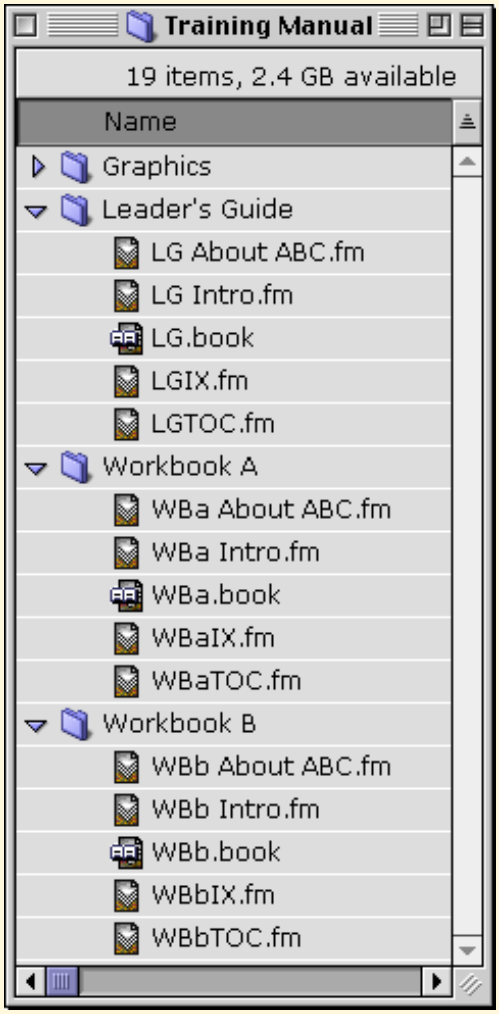
File	Leader's Guide	Workbook A	Workbook B
Book file	LG.book	WBa.book	WBb.book
Contents	LGTOC.fm	WBaTOC.fm	WBbTOC.fm
Chapter 1	LG Intro.fm	WBa Intro.fm	WBb Intro.fm
Chapter 2	LG About ABC.fm	WBa About ABC.fm	WBb About ABC.fm
Index	LGIX.fm	WBaIX.fm	WBbIX.fm

Organizing Files

There are basically two approaches to file organization, and the one you choose is a matter of preference.

Option 1: Use Separate Folders

This method is preferable because it clearly separates the files you work in (the leader's guide files) from the files you should not edit (the workbook files)

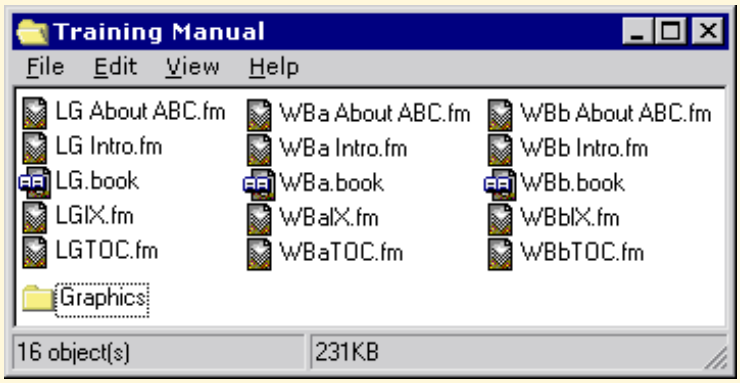


If you use this option, you could theoretically save yourself a lot of work by not renaming the files for either version of the workbook (just use different folder names). While this will work, it can be incredibly confusing.

- What you name the folders is not important. How you organize the hierarchy of the files is critical.
- In this example, Leader's Guide, Workbook A, and Workbook B should all be at the same hierarchical level in the file structure.
- If you use graphics imported by reference, Graphics can be anywhere in the file system except nested within Leader's Guide, Workbook A, or Workbook B.

If you nest Graphics or the workbook folders within Leader's Guide, any graphics imported by reference in the workbooks will be lost unless you keep an exact copy of the folder and its contents in each workbook folder (this is definitely not a good idea).

Option 2: Use a Single Folder



You may prefer to keep all of your files in a single folder. In this case, beginning each file name with LG, WBa, or WBb allows you to keep the three books visually distinct if you arrange your file view alphabetically.

If you use graphics imported by reference, this option has fewer restrictions. Since the files are all in the same folder, the relative path to Graphics stays the same no matter where it is. Graphics can go anywhere in your file system that you normally use for imported graphics.

Creating and Updating the Workbook

Introduction

Remember, you should not work directly in the workbook files. You will lose any work you perform in these files. All of your writing, editing and formatting should be done in the leader's guide files. You create the workbook files by renaming a copy of the leader's guide files and adjusting the conditional text settings for those files. There are two approaches to take:

- Create the workbook one document at a time
- Perform a batch conversion of the entire leader's guide into a workbook

One Document at a Time

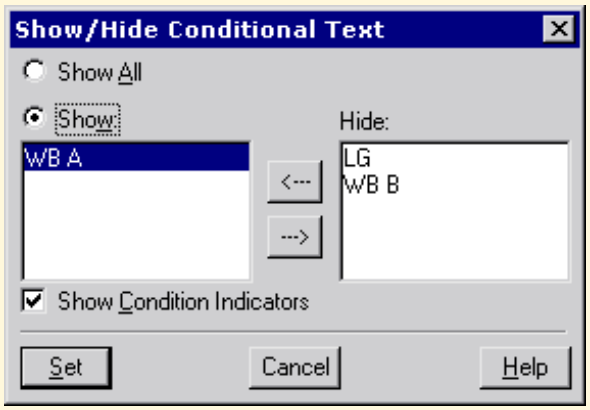
When you create a new chapter, or want to update just one chapter in the

workbook, it may be more expedient to create the workbook file individually rather than updating the entire book at once.

Create the WB File

1. Open the LG file.
2. Select File > Save As...
3. Navigate to the Workbook A (or Workbook B) folder.
4. Change the file name to match your naming convention.
5. Choose Save.
6. If asked if you want to replace or write over an existing file with the same name, select OK.

Change Conditional Text Settings



1. Select Special > Conditional Text...
2. Choose Show/Hide...
3. Add the LG and WB B conditions to the Hide list; leave the WB A condition in the Show list. (When creating Workbook B, the LG and WB A conditions should appear in the Hide list, and the WB B condition should appear in the Show list.)
4. Choose Set.

Update Cross-References

1. Select Edit > Update References...
2. Mark the All Cross-References checkbox.
3. Choose Update.
4. Select File > Save.

Create the WB B File

1. To create the WB B file directly from the WB A document, repeat the process beginning with step 2.

Update Cross-References in the LG

1. Open the LG file. If you have automatic updating of cross-references turned on, skip the next three steps.
2. Select Edit > Update References...

3. Mark the All Cross-References checkbox.
4. Choose Update.
5. Select File > Save.

Batch Conversion

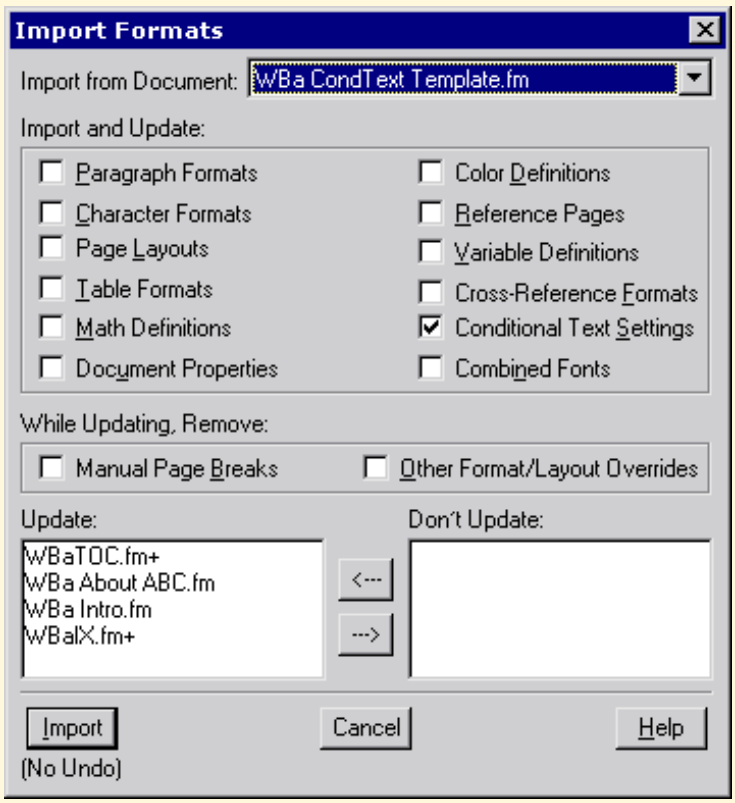
Although you may become quite skilled at going through the whole process of creating one workbook file at a time, it is quicker to update the whole book at once. This is the process you will need to go through every time you want to create up-to-date versions of the workbooks and update cross-references in the leader's guide.

Before you begin, you will need to create templates that have the Show/Hide properties set appropriately for each version of the workbook. Details about how to set up these templates, and additional suggestions for making the most of templates are covered in Importing Templates.

Copy and Rename the Documents

1. Delete any existing WB A documents (do not delete the book file).
2. Copy the LG documents (except the book file) into the Workbook A folder.
3. In the file system, rename the files to match your WB A naming convention.

Import Formats Across the Book



1. Open the WB A template.
2. Open the WB A book file.

If there are new files that have not yet been added to the book, add those files and make any changes necessary in File > Set Up File... now.

1. Select File > Import > Formats...
2. Set Import From Document to the WB A template.
3. Make your choices for what to Import and Update across the book (Conditional Text Settings must be checked).
4. Move all the files to the Update list.
5. Choose Import.

Generate/Update the Book

1. Select File > Generate/Update.
2. Make any adjustments necessary to the Generate list.
3. Choose Update.
4. Select File > Save.

Workbook B

1. Repeat the batch conversion steps for version B of the workbook.

Generate/Update the Leader's Guide

Always finish by generating the LG book in order to ensure that all cross-references are updated and resolved.

1. Open the LG book.
2. Select File > Generate/Update.
3. Make any adjustments necessary to the Generate list.
4. Choose Update.
5. Select File > Save.

***Continued in the next issue of* InFrame.**

Autonumbering in FrameMaker

Author: Dan Emory

Publication Date: 8/11/99

1 How Autonumbers are Defined

FrameMaker's autonumbering capability is quite versatile, allowing it to be used for many purposes. But versatility has a price—complexity. You must understand how autonumber formats are defined, and what their components are.

Autonumbers are defined as part of a paragraph format

The Numbering Properties panel of the Paragraph Designer is where you define an autonumber format.

Components of an Autonumber Format

An autonumber format can have any or all of the following components:

Series Label

A series label, if it is used, must be entered as the first component in the autonumber format text box of the Numbering Properties panel in the Paragraph Designer. The purpose of the series label is to differentiate different types of autonumber series within the same text flow. A series label consists of a single letter, followed by a colon. Here's an example of a series label:

```
H:
```

The series label does not appear in the displayed autonumber.

Note: Series labels are not required for unordered lists (e.g., bulleted lists), because sequential numbering is not involved

---

Any paragraph's autonumber within the same series is based on the previous numbered paragraph within that series. However, the same series can be used over and over again (e.g., numbered lists) within the same text flow, provided the first paragraph in each usage re-initializes the series numbering.

Counter Chain

A counter chain consists of one or more counters, where each counter produces either nothing or a single number or letter in the number series.

Counter

A counter is a placeholder that FrameMaker replaces with a single number or letter in a sequence. It is represented by opening and closing angle brackets (< >). A counter consists of one of the following building blocks:

<n=1> - Set the counter to a value of 1.

<n+> - Increment the counter by 1.

< =0> - Reset the counter to a value of 0, but do not display the value of 0 in the number of the paragraph.

<n> - Display the number of the most recent preceding paragraph in which the counter was incremented, set to 1, or set to 0.

< > - Ignore the value in this counter (i.e., do not include it in the number of the paragraph).

The letter that appears within the counter specifies the type of numbering, as follows:

- n = arabic numbering
- a = lowercase letters (a to z)
- A = uppercase letters (A to Z)
- r = lowercase roman numerals
- R = uppercase roman numerals.

Text, tabs, spaces, and punctuation

An autonumber can also include text, spaces, punctuation, or tabs anywhere in the autonumber format. Note that an autonumber format does not have to include a counter chain or a series label. That is, it can consist solely of text, tabs, spaces, and punctuation.

Note: When you use hyphens in autonumber formats it is advisable to specify a nonbreaking hyphen (dialog box code \+). This will prevent the number from being split at the hyphen in cross-references.

By the way…

When you specify `paranumonly` in a running header/footer variable, a cross-reference format, or a list specification flow, text, tabs, spaces, and punctuation appearing before or after the counter chain in the autonumber format do not appear in the header/footer, cross-reference, or generated list.

For example if you specify `paranumonly` in a cross-reference, here's what you'll get for the autonumber formats shown below:

Note: The dialog box code "\+" in the formats shown below specifies a nonbreaking hyphen..

FORMAT	THE CROSS-REFERENCE WILL CONTAIN:
Figure <n>\+<n+>:	**1-1** (the word Figure followed by a space at the beginning, and the colon and space at the end are excluded)
<n>\+< ><n+>: < >	**1-1:** (the empty counters are excluded, but the colon and space following the third counter are included because they are within the counter chain)
<n>\+< ><n+>< >:	**1-1** (the empty counters are excluded. Also, the colon and space at the end are excluded, because they are outside the counter chain)

Character Format

The Numbering Properties panel of the Paragraph Designer allows you to specify any character format in the character catalog for formatting the autonumber (default value is Default ¶ Font).

Position

The Numbering Properties panel of the Paragraph Designer allows you to specify that the autonumber appear at either the beginning or end of the paragraph (default is Start of Paragraph).

2 Autonumbering in Books

An autonumber format that has a Series Label can span all files of a book.

In other words, a series that begins in one file of a book can continue through succeeding files of the same book. For example, an autonumber series that specifies (among other things) the chapter number will number the chapters consecutively throughout the book. To get this behavior, proceed as follows:

Step 1. Open the book file.

Step 2. Select a file in the book where autonumbering is to be continued.

Step 3. Choose File > Set Up File. In the Set Up File dialog:

a. Set Paragraph Numbering to “Continue”.

b. If page numbering restarts at 1 within each chapter:

1. If the file is the first file in a chapter, set Page Numbering to “Restart at 1”, OR

2. If the file continues a chapter that began in a preceding file, set Page Numbering to “Continue”.

c. If page numbers are to be prefixed with the chapter number followed by a hyphen in generated lists and indexes. enter “n-” in the Prefix text box, where n = the number of the chapter.

Step 4. Repeat steps 2 and 3 for each file in the book.

With the setup made in step 2a, all paragraph number restart actions (e.g., restarting level 1 head numbering at 1 within each numbered chapter) are accomplished by resetting counters (i.e., < =0) in the counter chain.

3 Advantages of Using FrameMaker+SGML

FrameMaker+SGML can be used to create structured, as well as unstructured documents. In structured documents, an Element Definition Document (EDD) defines both the document’s structure and its formatting. The formatting information in the EDD can include all of the autonumber formats. This document was created as a structured document in which the EDD specifies all autonumbering formats.

The advantages of defining autonumber formats in the EDD include:

- Autonumbering is not tied to particular paragraph tags in the paragraph catalog. Instead, autonumber formats are tied to structural elements. The format rules for any such element can specify any required paragraph tag to be used for that element in each context. Other format rules for the same element can specify (usually via format change lists) the autonumbering format, if any, to be applied to the specified paragraph tag in each element context.

- In addition to structural context, format rules can specify the autonumbering format of an element based on attribute values in the same element, or in an antecedent element. Consequently, the autonumbering format applied to each element can be determined by a combination of structural context and attribute values.

What this all means is that any tag in the paragraph catalog can have applied to it any EDD-format-rule-specified autonumber format.

4 Some Examples

4.1 Autonumbering of chapters, headings, figures, tables, and equations

A single autonumbering series (series label H) can be used for this purpose, which prefixes headings, figures, tables and equations with the chapter number. Table 1 below shows the counter chain for each paragraph tag.

Table 1. Autonumbering format for the H series

Para Tag	Counter							Comments
	Chapter[abc]	Head 1	Head 2	Head 3	Figure No.	Table No.	Egn No.	
ChapTitle	**H:CHAPTER <n+>**	**< =0>**	**< =0>**	**< =0>**	**< =0>**	**< =0>**	**< =0>**	Increments Chapter counter, and resets all other counters to 0
Heading1	**H:<n>\+**	**<n+>**	**< =0>**	**< =0>**	**< >**	**< >**	**< >**	Increments Head1 counter and resets Head2 and Head3 counters to 0
Heading2	**H:<n>\+**	**<n>.**	**<n+>**	**< =0>**	**< >**	**< >**	**< >**	Increments Head2 counter and resets Head3 counter to 0
Heading3	**H:<n>\+**	**<n>.**	**<n>.**	**<n+>**	**< >**	**< >**	**< >**	Increments Head3 counter
FigCaption	**H:Figure <n>\+**	**< >**	**< >**	**< >**	**<n+>**	**< >**	**< >.**	Increments Figure No. counter. Note that the period that follows the figure number appears after the empty Eqn No. counter, so that it will be excluded when `paranumonly` is used in header/footer variables, cross references, and list specification flows.
TblCaption	**H:Table <n>\+**	**< >**	**< >**	**< >**	**< >**	**<n+>**	**< >.**	Increments Table No. counter Note that the period that follows the figure number appears after the empty Eqn No. counter, so that it will be excluded when `paranumonly` is used in header/footer variables, cross references, and list specification flows.
EqnCaption	**H:Eqn <n>\+**	**< >**	**< >**	**< >**	**< >**	**< >**	**<n+>.**	Increments Eqn No. counter
ChapNum	**H:Chapter <n>\+**	**< >**	**< >**	**< >**	**< >**	**< >**	**< >**	If chapters span two or more files, this tag is inserted in each file and used to create the chapter number prefix in the current page number variable (`paranumonly` is used in the variable definition to exclude the Chapter prefix). It can also be used in cross-references (in this case, `paranum` is used in the cross-reference format so as to include the Chapter prefix). The format for paragraph tag ChapNum specifies a default font of 2 pts, with color matching the background color, so it will be invisible.

a. H: is the Series label.
b. Note that the words Chapter, Figure, Table, and Eqn (and their following space character) are outside the counter chain, thus they are excluded when `paranumonly` is used in header/footer variables, cross references, and list specification flows.
c. The dialog box code "`\+`" produces a nonbreaking hyphen.

4.2 Outline-Style Autonumbering

Table 2 below shows the autonumbering series (series O) for creating outline-style autonumbering. Five levels are shown.

Table 2. Autonumbering for the O series

Para Tag[a]	Counter					Comments
	Level 1[b]	Level 2	Level 3	Level 4	Level 5	
Level1	**O:<A+>.**	**< =0>**	**< =0>**	**< =0>**	**< =0>**	Increments Level1 counter, and resets all other counters to 0
Level2	**O:< >**	**<n+>.**	**< =0>**	**< =0>**	**< =0>**	Increments Level2 counter, and resets all lower-level counters to 0
Level3	**O:< >**	**< >**	**<a+>.**	**< =0>**	**< =0>**	Increments Level3 counter, and resets all lower-level counters to 0
Level4	**O:< >**	**< >**	**< >**	**<n+>)**	**< =0>**	Increments Level4 counter, and resets all lower-level counters to 0
Level5	**O:< >**	**< >**	**< >**	**< >**	**<a+>)**	Increments Level5 counter

a. The paragraph formats for the Level2 thru Level5 tags have their First and Left indents set up to produce the desired amount of indentation for each level.
b. O is the series label.

4.3 A Short Centered Line

A short centered line can be created with the following autonumbering format specified for the CenteredLine paragraph tag:

\t\t

Where \t is the tab stop building block, and the tab stops are set in the CenteredLine paragraph format as follows:

Tab Stop 1 = TFW/2 - L/2 (a left-aligned tab stop).

Tab Stop 2 = TS1 + L (this left-aligned tab stop specifies a custom leader that uses the underline character).

Where:

TFW = the text frame width

L = the length of the line to be drawn

TS1 = the position of Tab Stop 1.

This produces the 1.375” centered line shown below:

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

FM+SGML Information Design

Author: Dan Emory,
Dan Emory & Associates

Although the FM+SGML Developer's Guide provides all the details needed to produce an Element Definition Document (EDD), it offers little guidance about information design methods. Nor does the Developer's Guide or any other Adobe-provided documentation discuss the complementary roles that the EDD and its templates play in implementing sound information designs. This paper is an attempt to remedy those deficiencies.

This paper (a structured document created with FM+SGML) focuses on the case where there is no existing SGML Document Type Definition (DTD), which provides the greatest opportunity for optimizing information design. However, many of the same techniques can be applied when there is a preexisting DTD. But a preexisting DTD often imposes a severe constraint on the design optimization methods discussed here.

1 Overview

*Note that the subject of this paper is **Information Design**, not document design..*

Marcus Carr defines where we're going

I correspond frequently with Marcus Carr of Allette Systems (Australia), who has broad experience with SGML and XML. In a recent email posted on a discussion list to which we both subscribe, he wrote:

> "…but what if the data also needs to be communicated to a database, not just another person? It's unreasonable to expect the database to understand n+ word processors…The only solution is to provide a common target for all applications that produce data, and build middleware to facilitate communication amongst them…"
>
> "As a technical communicator, you surely feel you are already a creator of information, but the goal that you're used to (making people understand) is being supplemented with other potentially more important imperatives. You will soon have to consider not only how to make information clear to a user, but also how to make it clear to whatever other applications may wish to make use of your data…"
>
> "…I'm unashamedly biased—if it was up to me, everyone would produce SGML data and we would dynamically bolt together fragments in response to specific queries…"

The New Paradigm creates new requirements for information Design.

The new paradigm described above will be implemented by object-oriented database repositories which directly store parsed SGML/XML and entities. Queries directed at this database will retrieve documents, or portions thereof, and middleware will dynamically assemble the retrieved components for delivery in the manner prescribed by the human user or the using application. The next five sections describe the elements of FM+SGML information design that I believe are essential to realizing the full potential of the new paradigm. Section 7 describes Extensible Markup Language (XML), which will be used to implement the new paradigm

2 Universality

Ideally, an enterprise would utilize a single EDD/DTD for all of its mission-critical document types: Everything from large multi-chapter manuals to standalone articles, specifications, proposals, training materials, and even memos, could be delivered either as printed documents, or as documents for on-line viewing in a variety of formats to meet different customer and departmental requirements.

HTML has proven the advantages of universality

HTML's huge success, despite its many shortcomings, demonstrates the overarching advantage of universality. The most salient feature of HTML is a simple, flexible structure in which the structural components are *document object types* not *information types*. Contrast this with the typical reductionistic SGML DTD, which prescribes a rigid structure of elements whose names describe information types rather than document objects.

The reductionistic approach to structure and element naming destroys universality

The reductionistic approach is doomed from the outset. Information content has many facets that cannot possibly be described in a single (usually cryptic) element name. This approach usually leads to the following undesirable results:

1. The DTD/EDD is specific to a particular document type, thus its universality is destroyed.
2. The DTD/EDD is volatile because it is susceptible to change every time there is a shift in technology or processes. These shifts often create new information types, and make others obsolete, requiring the addition of new elements and the deletion of others. The resulting volatility invalidates legacy documents prepared to the earlier version of the DTD/EDD.
3. Either the element names are so generic that they fail to supply useful information about the content,

   OR

   The number of elements expands to describe each possible combination of information facets, making the DTD so unwieldy, reductionistic, volatile, and arcane that no one can use it.

Note: A reductionistic dimension to information design is revived in Section 5, *Extensibility of the Modular Structure*.

Attributes should define information types

The obvious way out of the reductionistic dead-end is to use attributes, rather than element names, to describe the information content of each element. A set of generic attributes, easily adaptable and extensible for different organizational requirements, can provide the multiple facets needed to adequately describe information content.

Element names should describe document object types

Authors who create documents with word processors and desktop publishing systems share a common way of thinking about document objects (e.g., headings, paragraphs, strings, lists, steps, notes, cautions, warnings, graphics, equations, tables). When an author looks at an element catalog listing all the valid elements that can be inserted at some point in a document structure, s(he)'s not thinking about information content, s(he)'s thinking about document objects. Authors need to know what kind of document object will be created by each element so they can choose the appropriate one.

Requiring authors to choose among elements whose names describe information types rather than document objects is like telling a traveller he must first reach his destination before he is permitted to choose his mode of travel. It's counterintuitive and counterproductive.

The best of both possible worlds

Using document objects for element names and attributes to describe information content provides the best of both possible worlds. Major benefits of this approach include:

Benefits Derived from a Universal DTD/EDD

- **Information Reusability**: Since all document types utilize the same DTD/EDD, information "packets" extracted from any document can be reused in any other structured document, simply by copying and pasting them.
- **Reduced Training Costs**
- **Reduced Operating Costs**
- **Less Volatility**: Since the set of document object types is relatively stable, DTD/EDD volatility can be greatly reduced when element names describe document objects.

3 Structural Enrichment

A high-end DTP for unstructured documents resembles a Lego set, where the immutable set of building blocks is a relatively small group of basic document object types. In FrameMaker, a template, consisting of a set of catalogs, adds value by defining all of the different document object subtypes needed by authors. Using these catalogs, authors can string pre-formatted document objects together in any sequence or combination allowed by the DTP.

Fundamentally, the purpose of a DTD/EDD should be to impose order and consistency on the document structure, without limiting the range of options needed by authors to present complex information. We need an orderly way to *enrich the structure when required*.

An example of structural enrichment

Take for example, a list of numbered steps for a procedure:

Step 1. Some steps may require multiple levels of autonumbered substeps:

a. Substep a

   1. First sub-substep.
   2. A second sub-substep.

b. Substep b.

Step 2. A step may have multiple paragraphs, where only the first paragraph is numbered, as demonstrated below:

The succeeding paragraphs are indented to the same level as the text in the numbered paragraph so as to make clear it is part of the step.

Step 3. In some cases, a step needs to be preceded or followed by a a note, caution, or warning which is an integral part of the step:

Note: This is a note that precedes (and pertains to) Step 4. The `Alert` container element prescribes the note's structure and format. This note is actually part of the `Step` element containing the text of Step 4.

Step 4. In other cases, a step may include a graphic, table, or other object which is inserted below the step's text, as demonstrated below:

By the Way...

This is a 1-row, 1-column table containing special instructions or explanations pertaining to a procedural step.

Methods for achieving structural enrichment

The DTD/EDD structure rules for the example shown above should allow all of the indicated possibilities, and more. There are two methods for accomplishing this:

1. List all of the possible object types shown above as inclusions in the `Steps` container (a numbered list), whose general rule is:

   ```
   Step, Step+
   ```

   This is the easiest solution, but it is also the least desirable one, because it clutters up the element catalog with those inclusions, which is confusing to authors because the inclusions will be indicated as valid anywhere in the `Steps` container, even though they would all be invalid everywhere except at the end of a `Step` paragraph.

2. Include in the structure rule for each `Step` element one or more optional **"structure enricher"** container elements. Let's call one of them `#Body`, where the "#" prefix indicates that it is optional, and the other `Alert`, which creates a note, caution, or warning before the text of the Step element. The #Body element is an "attachment spine" for miscellaneous document objects. The General Rule for the `Step` element specifies that the `#Body` container can optionally be inserted at the end of the text in any numbered `Step` container element, and that an `Alert` container element can be optionally inserted before the text of the step. That is, the general rule for element `Step` is:

   ```
   Alert?, <TEXT>, #Body?
   ```

   and the general rule for the `#Body` element might specify:

   ```
   (Figure | Next_Level | Table | Equation | More_Text)*
   ```

   Where:

   - `Figure` produces a captioned or uncaptioned graphic
   - `Next_Level` produces substeps under a numbered step, or sub-substeps under a substep.
   - `Table` produces a table of any selected type
   - `Equation` produces a captioned or uncaptioned equation
   - `More_Text` produces additional unnumbered text paragraphs under a numbered step or substep paragraph.

Unlike inclusions, the element catalog will indicate that the #Body container is valid only when the text cursor is placed at the end of a Step paragraph, and that the Alert container is valid only before the text.

If the author needs one or more of the object types provided by #Body, its insertion at the end of a Step paragraph causes the element catalog to list all of the object types included in the #Body container's general rule.

This method has the added advantage of "encapsulating" all of those added objects as part of the Step element to which they pertain. Consequently, when such a Step element is moved or reused, *all of its attached objects go with it*.

Figure 1 shows the structure view of the implementation described in item 2 above, as it would be applied to the four-step example presented earlier in this section.

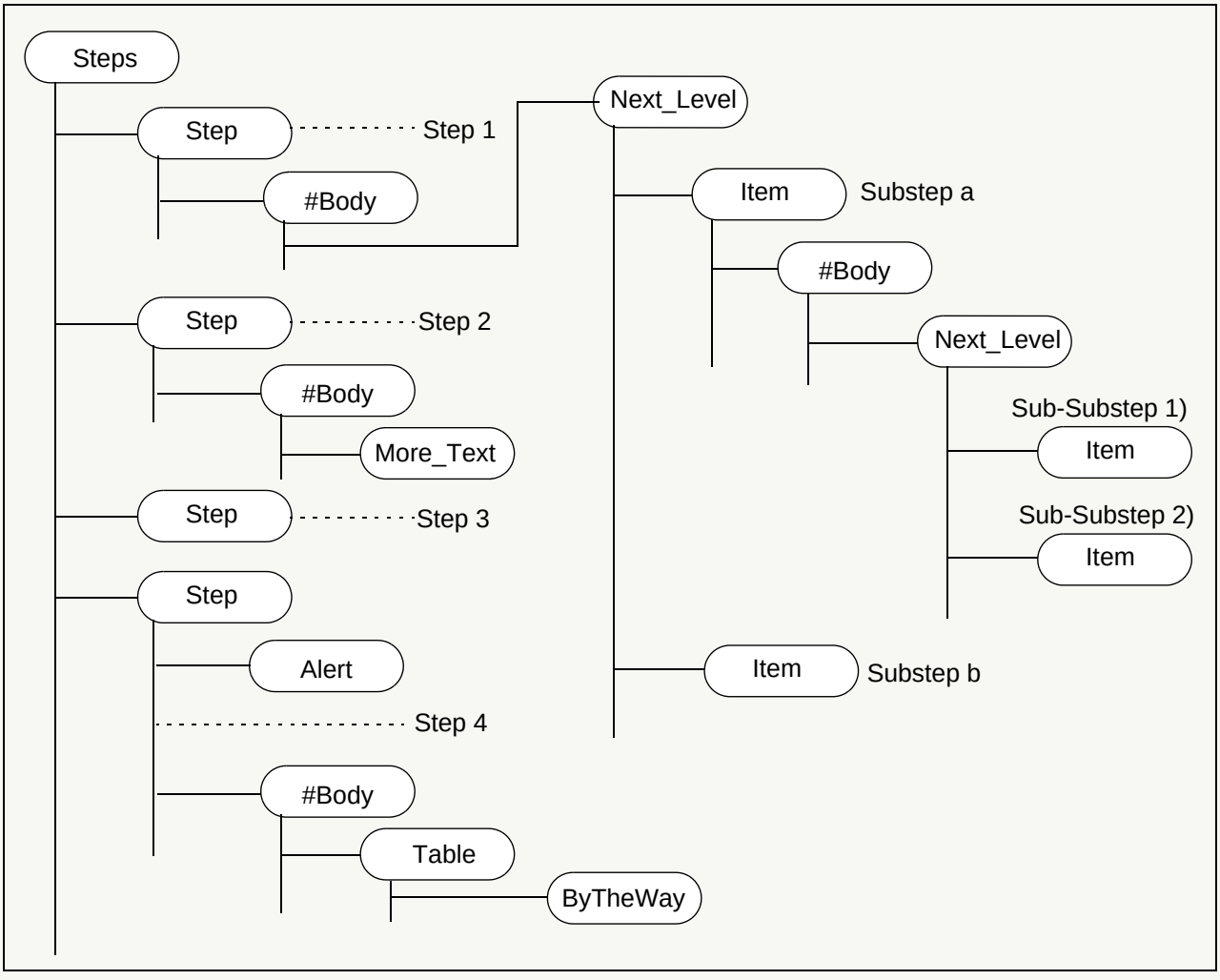


Figure 1. Structural Enrichment Example

4 Encapsulation

Encapsulation provides an effective way of managing, reusing, retrieving, and delivering packets of information below the document or file level. "Wrapper" elements that provide the means of encapsulation should offer ***metadata enrichment*** *(i.e., additional attributes) that enhance the ability to manage, reuse, and retrieve the encapsulated information.*

Moreover, encapsulation wrappers can greatly facilitate collaborative authoring, where two or more individuals work simultaneously and without conflict on different parts of the same document .

4.1 Uses of Encapsulation Wrappers

Creating Text Insets

If an encapsulation wrapper is valid at the highest level, it can be used to create text insets of the following types:

- Create libraries of repetitively used boilerplate material, such as standardized notes cautions and warnings, contract clauses, etc. By creating each such boilerplate packet in a separately named text flow within a single FrameMaker file, an entire library of boilerplate material can be contained within a single FrameMaker "fragment" file.
- In the same manner described above, each author in a collaborative authoring environment can create his/her individual contributions to a writing project in separately named text flows within a single FrameMaker fragment file. When used in this manner, the master document files that use the individual contributions essentially become "skeletons".

Any separately named text flow from a fragment file can then be added as a text inset to any document that uses the same EDD, as follows:

1. Insert an `Inset_Wrapper` element whose structure rules include the encapsulation wrapper used in the fragment files. The `Inset_Wrapper` element has attributes that identify the name of the source fragment file, the source fragment within that file and any other needed amplifying information (e.g., the author's name).
2. Once the `Inset_Wrapper` element is inserted, any selected text inset from a fragment file can be imported by reference as a child of `Inset_Wrapper`.

After these steps are taken, all text insets in a document are automatically updated whenever their source fragment is updated in its fragment file.

Encapsulation of information packets within documents

Candidates for encapsulation include:

Minimum Deliverable Units (MDUs)

An MDU would typically be a single subject (e.g., a section) within a document. The section and all its subsections constitutes an information packet suitable for independent delivery for on-line viewing or other purposes. Delivery of MDUs stored in a database could result, for example, from the outcome of a search for some combination of attribute values in MDU encapsulation wrappers. Only the encapsulated MDUs which satisfy the search criteria are delivered. This delivery approach allows users to find and analyze relevant information much more rapidly than if an entire document were delivered for each database "hit".

Minimum Reusable Units (MRUs)

An MRU is a group of contiguous elements that forms a reusable information packet. Delivery of MRUs could result, for example, from the outcome of a database search for some combination of attribute values in MRU encapsulation wrappers. Such queries would allow authors to rapidly determine which MRUs, if any, match the requirements for for a specific reuse.

4.2 How Encapsulation Wrappers are Implemented

Structure Rules

Encapsulation wrappers are special attachment spines that are distinguished from "ordinary" attachment spines by their different purpose. The structure rule for an encapsulation wrapper is nearly identical to the structure rule for the "ordinary" attachment spine that contains it. Consequently, any subset of contiguous elements attached to an "ordinary" attachment spine can be encapsulated.

But these same encapsulation wrappers are also valid at the highest level so that they can be used to create text insets in separate text flows of a fragment file. Figure 2 shows an implementation.

Metadata Enrichment

Attributes provide the way for encapsulation wrappers to provide details about the information content, as well as information needed to properly manage and reuse the data. Here is a representative list of the attributes which might be included in encapsulation wrappers:

- ResourceID: An Identifier (e.g., a Universal Resource Identifier) that uniquely identifies the resource represented by the MDU or MRU.
- ProjectID: Identifies the project under which the encapsulated data was created.
- WorkOrder: Identifies the work order that authorized the creation of the encapsulated data.
- Subject: The subject(s) of the encapsulated data.
- Description: More detail about the content of the resource.
- Purpose: Describes the purpose of the information.
- InfoType and InfoSubtype: These attributes Identify the information type and sub-type.
- UsedIn: Identifies the document for which the encapsulated data was originally created.
- Effectivity: Indicates product release numbers, model numbers, etc. for which the encapsulated data is valid.
- ECOs: Identifies Engineering Change Orders that have impacted the content of the encapsulated data.
- Author: Identifies the author of the encapsulated data.
- Owner: Identifies the owner (e.g., a department or project) that has control over the content of the encapsulated data.
- SecurityClass: The security classification of the encapsulated data.
- Keywords: Lists the applicable keywords/phrases. Allows keyword searches to be conducted on attribute values rather than text, guaranteeing that all hits are relevant.
- Correlations: One or more attributes identify design documents, spec-

ifications, regulations, etc. that influenced the information content. For example, an OSHA regulation may prescribe how a caution or warning is written. If the regulation changes, an attribute search for the regulation number yields all cautions or warnings that might be affected by the change.

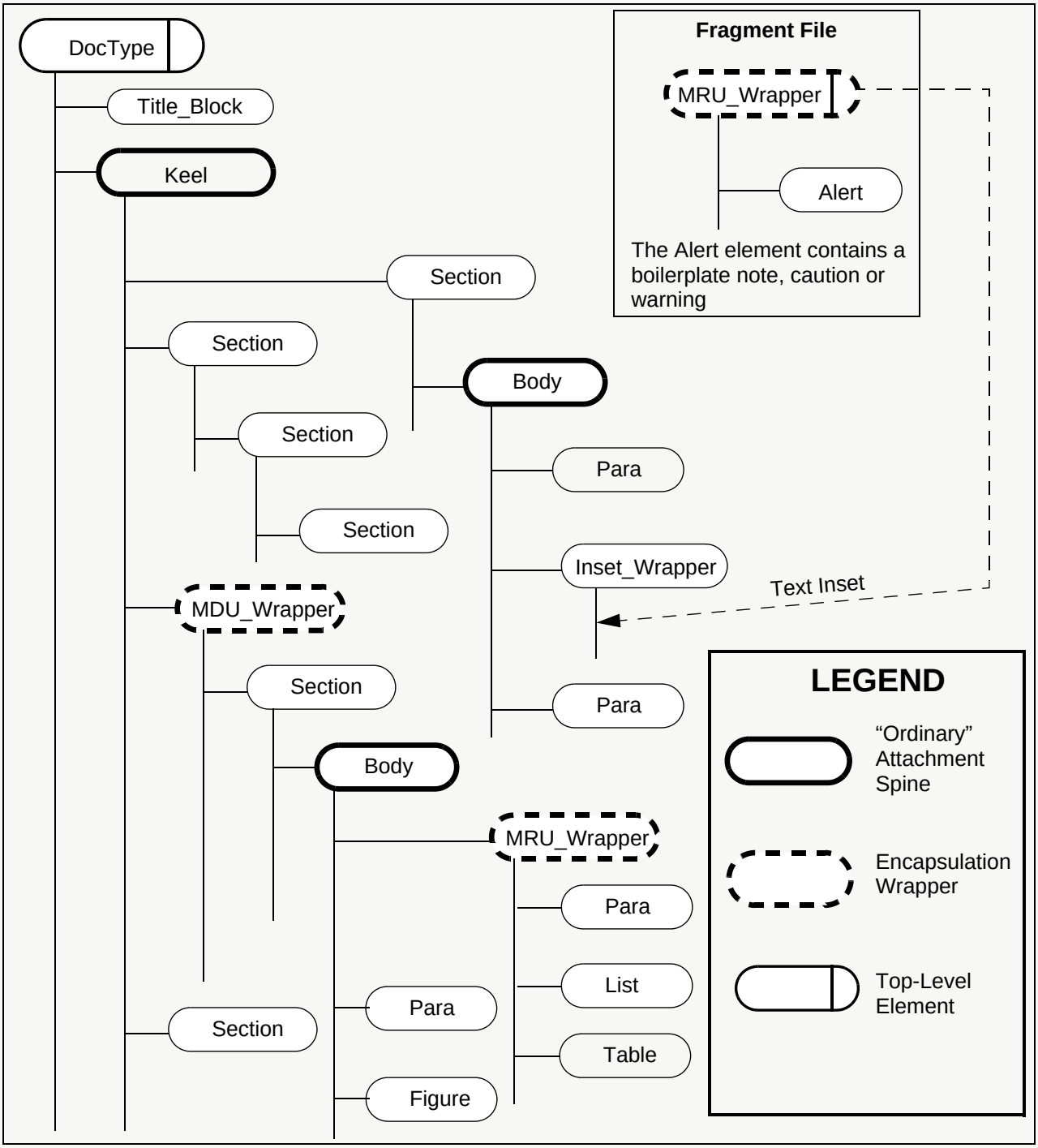


Figure 2. An Encapsulation Wrapper Implementation

4.3 Low-Level Encapsulation

*The information packets (hereafter called **chunks**) within each titled section should also be titled and encapsulated.*

An example of a multilevel chunk structure (Level 1)

This is an example of a multilevel chunk structure. At each level but the last, the title appears in the sidehead, and all of the document objects (text, graphics, tables, etc.) that are part of the chunk appear in the normal text column. Each chunk element has an optional #Body attachment spine for attaching such document objects.

Level 2 Chunk

Formatting differentiates a second-level chunk from a first-level one.

Level 3 Chunk

Indenture and formatting differentiate a third-level chunk from a second-level chunk.

Level 4 Chunk

Formatting differentiates a fourth-level chunk from a third-level chunk.

Level 5 Chunk: The fifth-level chunk appears in the normal text column, with the title paragraph runin with the text paragraph that follows.

Chunk structures provide a simple way to title and encapsulate information packets below the section level

Figure 3 is the structure view of the 5-level structure shown above. Notice that the Chunk_Level1 element encapsulates the entire structure. It should also be observed that chunk structures are natural candidates to become MRUs, in which case their metadata can be enriched by wrapping them in an MRU_Wrapper.

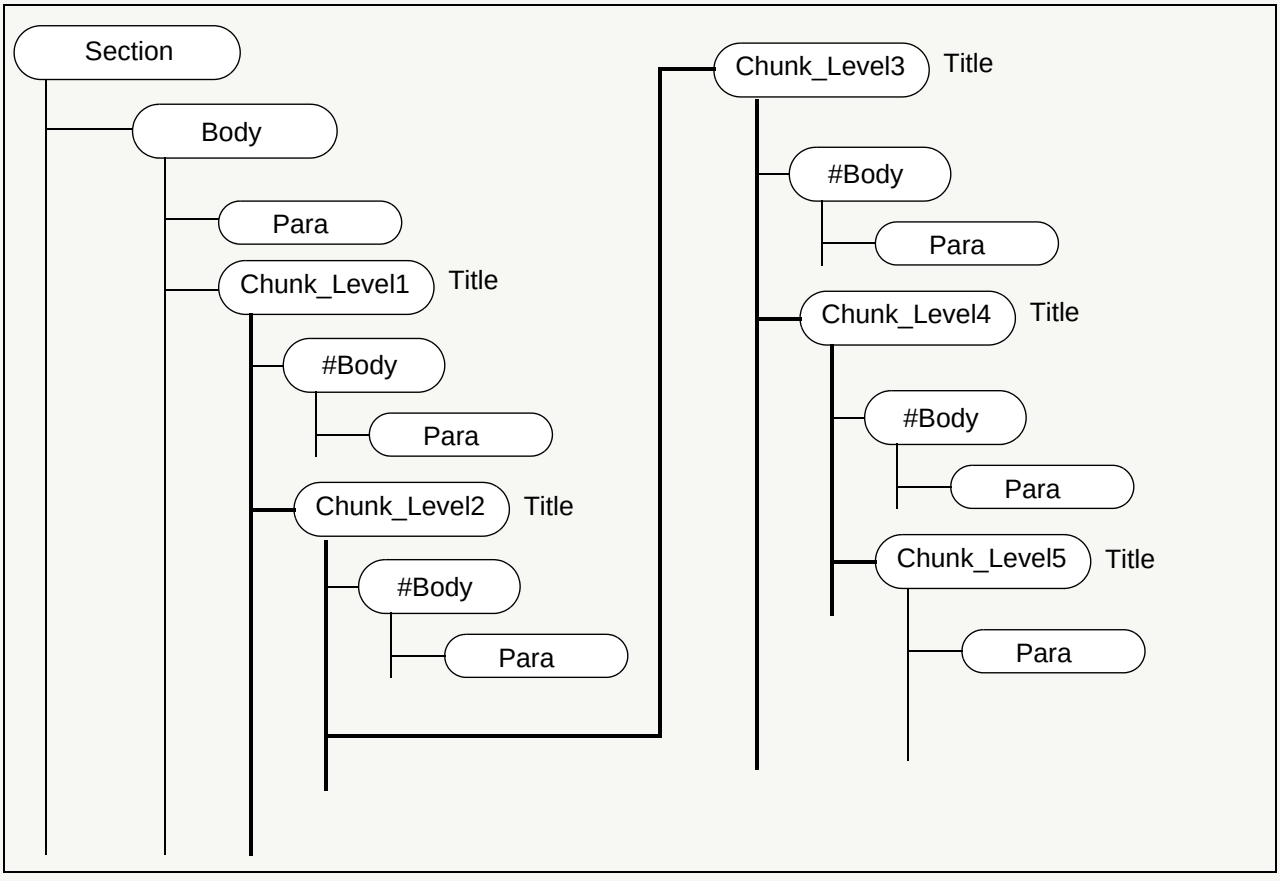


Figure 3. Chunk Structure Implementation

5 Extensibility of the Modular Structure

Having demolished the reductionistic approach to DTD/EDD design in Section 2, I now resurrect it.

The Modular Design Concepts Presented Thus Far

The modular design presented in Sections 3 and 4 includes:

1. Three levels of "Ordinary" attachment spines, including:
   a. `Keel`, which is the attachment spine for first-level section heads and MDU encapsulation wrappers.
   b. `Body`, which is the mandatory attachment spine, or "sub-keel", for chunk structures, miscellaneous document objects, and MRU encapsulation wrappers under section heads.
   c. `#Body`, which is the optional "structural enrichment" attachment spine for attaching miscellaneous document objects and MRU wrappers under chunk structures, items, paragraphs, and other text container elements.
2. Encapsulation wrappers, which include:
   a. Wrappers that are valid at the highest level, which are used to create text insets in fragment file text flows.
   b. Wrappers for Minimum Deliverable Units (MDUs), which can be attached to the `Keel` attachment spine, or to first- and second-level section heads.
   c. Wrappers for Minimum Reusable Units (MRUs), which can be attached to the `Body` and `#Body` attachment spines.
3. Titled Chunks, attached to the `Body` attachment spine, which serve as encapsulators of low-level structure.

Figure 4 presents a generalized structure view showing how documents are constructed from these modular components.

The main advantages of a modular structure are its extensibility and flexibility

Suppose, for example, the DTD/EDD defines many different document types, and each such type requires various types of MDU/MRU-type wrappers whose names describe doctype-specific information content (e.g., the names of the major structural components of an automobile or aircraft). This could be easily done, as follows:

1. Expand the structure rules for the `Keel`, `Body`, and `#Body` attachment spine elements to include the names of the additional MDU- and MRU-type wrappers that can be attached to them.
2. In the structure rule of the top-level element for each doctype, insert an exclusions line that lists those MDU- and MRU-type wrappers that are not applicable to that doctype.

The structure rules for each doctype-specific encapsulation wrapper could define structures and elements that are unique to that doctype.

If this approach were taken, information interchange and universality could be preserved by adding a "Generic" doctype whose top-level element does not have any exclusions, thus all MDU- and MRU-type wrappers and their children would be allowed in the "Generic" doctype.

It's also possible to define doctypes without section heads. The structure rule would omit the `Keel` element, replacing it with a `Body` element, which allows titled chunks to be substituted for section heads.

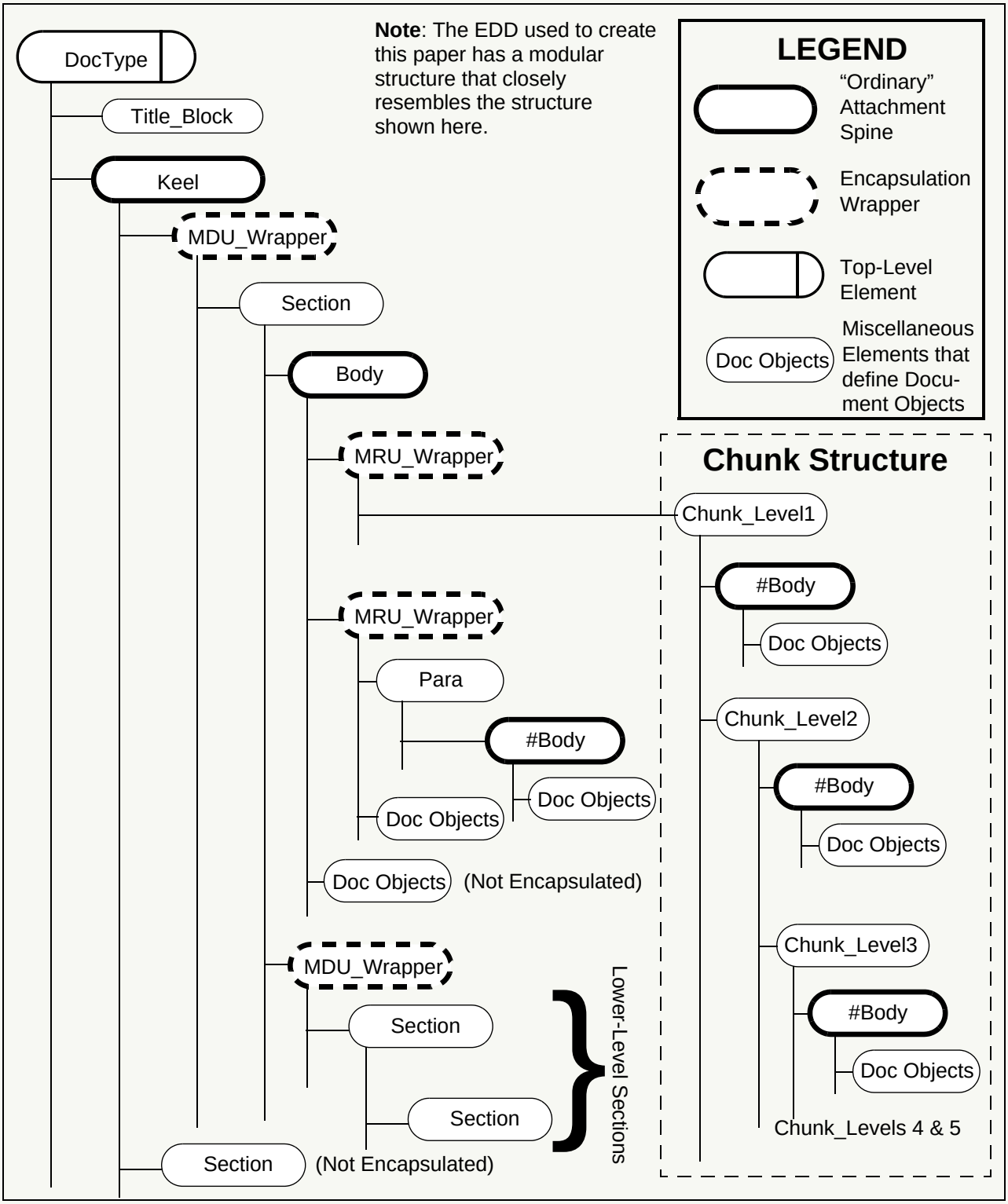


Figure 4. The Modular Document Structure

6 Using FM+SGML's Unique Feature Set

FM+SGML's feature set makes it well-suited for authoring structured documents, and as a print engine for outputting them to paper or PDF.

6.1 Advantages of FM+SGML as an Authoring and Formatting Tool

The authoring tool must provide authors with a WYSIWYG view

This statement simply affirms the basic premise underlying all high-end DTPs, namely that authors cannot effectively design how information is presented unless the on-screen editing view is an accurate representation of how the information will appear when it is read by the end user.

Authoring and formatting features of FM+SGML

- Its capability, through templates and EDD format rules, to format structured documents is unmatched
- Its WSYWIG editing view assures that authors can effectively utilize those formatting capabilities
- Its interactive structure view serves as a powerful editing tool, as well as an aid to the analysis and management of structure
- Its element catalog shows authors which elements are valid at any given insertion point.
- Its built-in validation capability detects the following types of anomalies:
  - — Invalid elements
  - — Missing elements that are required
  - — Invalid attributes
  - — Invalid values in attributes
  - — Missing values for required attributes.

  FM+SGML stops at each anomaly, highlights it in the structure view, and describes the defect
- Built-in Find/Change capability for searching on/changing, element tagnames, attribute names, and attribute values
- FM+SGML eliminates most requirements for format-related processing instructions (PIs) in SGML document instances.

Graphic Conversion

Graphics in FM+SGML-created structured documents can be exported to SGML as entities in almost any read/write rules-specified graphic format.

Text Insets

FM+SGML's text inset feature provides an innovative way to facilitate information reuse and collaborative authoring. Each structured text inset is encapsulated by an encapsulation wrapper element in a separate text flow within a fragment file.

6.2 FM+SGML's Formatting Capabilities

Formatting nuances help readers to comprehend complex information, and to rapidly scan through documents in search of the particular information they seek. Format variations are likely to be needed for different document types, different departmental standards within the enterprise, and different modes of information delivery.

The printed or PDF version of a document usually has the most demanding formatting requirements

It follows, therefore, that the combination of the EDD and its companion template(s) must be capable of satisfying the formatting requirements for high-quality printed or PDF output.

When this goal is achieved, it is reasonable to assume that the information design will also be adaptable to the formatting requirements for alternative (and less demanding) information delivery modes.

6.2.1 The Formatting Role of the EDD

The EDD prescribes how document objects are formatted.

The EDD's format rules define the format tagnames that will be included in the template's catalogs

When element definitions are imported into a template, the import action creates the EDD-specified tagnames in the template's catalogs. The affected catalogs are:

- The paragraph catalog, which contains the base paragraph format tags specified in the format rules of container elements.
- The character catalog, which contains the character format tags specified in the format rules of text range elements.
- The table catalog, which contains the table format tags specified in the format rules for table elements.
- The cross-reference format catalog, which contains the cross-reference tags specified in the format rules for cross-reference elements.

The EDD defines Format changes that are applied to the base paragraph tags in the template

These format changes are specified in All Context, Context, Level, Prefix, and Suffix rules of individual container elements.

6.2.2 The Formatting Role of FM+SGML Templates

An FM+SGML template is used to create new structured documents, or to modify the formatting of an existing structured document. Many templates may be created for the same EDD, each having format variations for different document types and/or different information delivery modes.

1. **The template specifies the formatting details of the tags added to its catalogs when the EDD is imported into it**

When these tags are created in the catalogs by the Import Element Definitions action, they are plain-vanilla. Formatting details must now be added by the template designer to fulfill the intended purpose of each tag.

2. **The template has nearly total control over many formatting aspects**

Among the aspects that are under near-total control of the template are:

- **Page Layouts**, including running headers and footers
- **Character formats**. In this case, the EDD format rules for text ranges specify "Use Character format", which creates the character tags in the character catalog when the element definitions are imported into the template. The template designer has control over the formatting details for each such character tag.
- **System and User Variable Definitions**

- **Color Definitions**
- **Reference Frames** on reference pages
- **Conditional Text Settings**
- **Math Definitions**.

3. **The extent to which a template can influence paragraph formatting is determined by how the EDD's format rules are written**

The Three EDD design methods described below demonstrate how the EDD's format rules can affect the extent to which templates influence paragraph formatting.

3.1 The "Top-Down Inheritance" Method

When this method is employed, elements that are valid at the highest level specify a base paragraph format (e.g., Body). Each container element descendant of that top-level element inherits the Body paragraph format, plus any antecedent format changes to it it which are not overridden by the element's own format rules. Consequently, any template created from such an EDD will have a single paragraph tag named Body in its paragraph catalog. Obviously, the influence of such a template on paragraph formatting is nil.

3.2 The "Use Paragraph Format" Method

When this method is employed, all container element format changes are in rules that specify "Use paragraph format". This was the only method available in FM+SGML's predecessor, FrameBuilder. In this case, the template has virtually total control over paragraph formatting.

3.3 The "Middle-of-the-Road" Method

Both of the two previously described methods have serious disadvantages. The middle-of-the-road method specifies a base paragraph format in the format rules for most container elements having <TEXT> in their general rule, so as to stop all antecedent format inheritance. The format rules required to modify the base paragraph format for each element context do not, in most cases, specify "Use paragraph format". Instead, they specify "Use format change list" (more about the advantages of format change lists in the next section).

Since many different container elements can specify the same base paragraph format, the huge proliferation in paragraph tags produced by the "Use Paragraph Format" method is avoided. Instead, a relatively small set of base paragraph formats is defined to accommodate variations in such formatting parameters as font family, font size, line spacing, table cell parameters, hyphenation, and Frame Above/Below for different types of container elements. Although there may be some exceptions, the EDD's format change lists generally do not override these template-defined formatting parameters, thus the template designer has control over them, making it possible to develop multiple templates for the same EDD in which the same set of base paragraph formats is formatted differently for different document types and/or different information delivery modes.

6.2.3 Advantages of Using Format Change Lists

When format change lists are utilized, virtually all EDD format rules for modifying base paragraph formats specify:

```
Use Format Change List: Name
```

Where "Name" is the name of a particular format change list.

Format Change Lists are modular formatting building blocks

Format change lists should be viewed as modular formatting building blocks that can be combined in many different ways to produce different formatting outcomes (e.g., the formatting for a particular element context can be specified in the element's All Context and Context rules to be the composite of two or more format change lists).

The Impact on EDD Design

Format rules in an EDD refer to format change lists, where the formatting details are provided. Since many format rules in many different elements can use the same format change lists, the volume of formatting details in the EDD can be greatly reduced, providing better management of the formatting problem, as well as making it easier to modify the EDD. In a modular EDD design, format change lists can facilitate reuse of structural modules between different EDDs.

Format Change Lists can provide even greater format adaptability

All of the format change lists in an EDD can be grouped together into categories at the end of the EDD. This makes it possible, for example, to easily clone multiple versions of an EDD, all of which have identical element names, structure rules, and format rules, plus the same set of named format change lists. However, the formatting details in those format change lists can vary from version-to-version so as to accommodate wide variations in format for different document types and delivery modes.

Suppose, for example, two versions of the EDD, each having its own template, are created, where one version is intended for producing printed documents, and the other version is used for producing PDF documents for on-line viewing. Each EDD is identical, except for the formatting details in their format change lists. These variations define the formatting differences between the two delivery modes. Further variations (e.g., differences in font family and font size) may exist in the formatting of the base paragraph set in the two templates.

Suppose further that you create the printed version first. Now, you want to produce the on-line version. All you have to do to accomplish the conversion is import into the document the formats and element definitions from the "on-line" template. Consequently, the same document files can be used to produce both the printed and on-line versions, even though they may differ widely in their formatting details.

6.2.4 Using Attributes to Provide Authors with Formatting Options

Without the use of formatting attributes, a container element's paragraph format is determined solely by its context within the structural hierarchy. If formatting attributes are not provided and different formatting options are required for the same basic element, multiple versions of that element (each with a different name, the same structure rules, and different format rules) must be created. This unnecessarily complicates the EDD, as well as the authoring task.

Choice-type formatting attributes provide an excellent way to avoid element proliferation, and can provide authors with the options they need to

optimize the presentation of complex information.

Formatting Attributes for a Para element

Take, for example, the ubiquitous Para element that serves as the general-purpose text container in many DTDs/EDDs. Here are some of the formatting options an author might like to have for the Para element:

1. Select the horizontal alignment of the paragraph as Left, Right, Center, Justified, or aligned on a decimal point.
2. Select the table cell vertical alignment of the paragraph as Top, Middle, or Bottom
3. Specify the amount of indenture of the paragraph from the left margin (e.g., Level 1, Level 2, etc.)
4. Change the font size of the entire paragraph from Regular (the default font size) to 2 points larger (Large) or 2 points smaller (Small) than Regular, with a corresponding change in the line spacing
5. Change the font of the entire paragraph to Courier or some other special font to represent, for instance, a computer message or a typed command
6. Make the style of the entire paragraph bold, italics, or underlined
7. Make the paragraph span all columns, both the sidehead and normal column, or only a single column
8. Force the paragraph to appear at the top of a column (Column Break) or at the top of a page (Page Break).

All of these options can be readily provided by choice-type attributes, in which the default value for each attribute produces the format specified by the applicable base paragraph format in the template.

Formatting Attributes at the Highest Level

Formatting attributes can also be used at the highest level to provide formatting options for an entire document. For example:

1. Autonumbering options for chapter and appendix titles (e.g., None, Alpha, or Number), where, if the title is numbered or lettered, the number or letter is prefixed to the section head numbers, if any.
2. Autonumbering options for section heads (e.g., None, Number Major Sections Only, or Number All Sections).
3. Section head styling options needed for different document types or delivery modes.
4. Autonumbering options for chunk structures below the section level, allowing titled chunks to be used, for example, as additional numbered section levels.
5. Figure, Table, and Equation autonumbering options (e.g., Restart at Chapter, Restart at Each Major Section, or Number From Start of Book), where:
    a. If the first option is chosen, numbering is restarted at 1 in each chapter or appendix, and, if the chapter or appendix title is autonumbered, that autonumber is prefixed to the figure, table, and equation autonumbers.
    b. If the second option is chosen, numbering is restarted at 1 in each major numbered section, and the section number is prefixed to the figure, table, and equation autonumbers.

c. If the third option is chosen, there is no prefix to the figure, table, and equation numbers, and they are numbered consecutively from the start of the book.

Implementation in the EDD

Formatting attributes for elements provide an effective way to give authors a wide range of formatting options without a concomitant proliferation in the number of elements in the EDD. Moreover, by using formatting attributes, EDD modifications to accommodate additional formatting requirements are easily implemented, either by adding new attributes, or by adding more choices to existing attributes, thus the DTD/EDD structure rules are unaffected.

Here is a typical format rule for a Style attribute in a Para element:

```
If context is: [Style = "Bold"]
   Use format change list: Bold
Else, if context is: [Style = "Italics"]
   Use format change list: Italics
Else, if context is: [Style = "Underline"]
   Use format change list: Underline
Else if context is: [Style = "Normal"]
   Use format change list: Normal
```

Note that many different container elements could have a Style attribute, and all would reference the same format change lists in their identical format rules for the Style attribute.

6.3 Style Guide Enforcement

An EDD's format rules constitute an auto-enforced style guide, freeing authors from concerns about style guide compliance. If authors attempt to override the format rules by applying unstructured character formatting within paragraphs, or by making *ad hoc* changes to paragraph formats, those overrides can be removed. This is accomplished by re-importing the document's Element Definitions (with Remove Format Overrides turned on). This action restores the affected document to full compliance with the EDD's format rules, including removal of all *ad hoc* character and paragraph formatting.

6.4 FM+SGML's Utilities and Developer's Tools

The following utilities and tools are are built into FM+SGML:

- **Batch Conversion Utilities** to convert FM+SGML documents to SGML, or to convert SGML documents to FM+SGML
- **Generate Structure Rules Tables** for an unstructured document, and then convert it to a structured document using those structure rules
- **Generate and Apply Paragraph and Character Format Tags** to a structured document in preparation for mapping those tags for an HTML conversion
- **Create a new EDD**
- **Create an EDD** from an existing DTD
- **Open a DTD** for editing

- **Create a DTD** from an existing EDD
- **Parse an Existing SGML Document Instance**, and log all detected anomalies
- **Edit the SGML import/Export application file**
- **Create a new Read/Write Rules file** for SGML import/export
- **Check an Existing Read/Write Rules file**, and log all syntax errors.

6.5 **Customization**

FM+SGML's Customization capabilities include:

- Customizing menus
- Customizing graphic filters
- Developing API clients with the Frame Developer's Kit (FDK)
- Developing SGML Import/Export Applications for accomplishing conversions between SGML/XML and FM+SGML
- Using FM+SGML 5.5.6's ODMA interface to create bridges to ODMA-compatible database repositories. Such bridges could allow FM+SGML to:
  - — Check SGML/XML documents (or portions thereof) out of the database, and import them into FM+SGML for editing
  - — Check FM+SGML documents (or portions thereof) back into the database as SGML/XML.
  - — Query the database
  - — Navigate around in the database.

7 XML for the New Paradigm

Extensible Markup Language (XML) will revolutionize the way in which information is handled and processed. It promises to make information "smarter" by including machine-readable data about the structure and content of information objects (hereafter called "resources"). Resources can include documents, document fragments, and external entities such as graphics and individual database records. Resources are always identified by Universal Resource Identifiers (URIs), plus optional anchor IDs. The extensibility of URIs allows the introduction of identifiers for almost any resource imaginable.

Note: Some of the text in this section paraphrases information that was originally contained in "*A New Dawn*", an article about XML by Glyn Moody.

XSL Stylesheets

It is a basic rule of XML that content and presentation are separate, thus XML tags contain no hint about how the information contained therein should be formatted/displayed. One candidate for creating stylesheets uses an XML language called eXtensible Style Language (XSL). XSL stylesheets format the data for display or printing, but also promise much more. For example, different stylesheets could be applied to the same data; each stylesheet could hide some information chunks, and display others.

Hypertext Links

Another XML application, called XLink, makes hypertext links much more robust than they are in HTML or PDF. XLink offers a number of new features such as links that indicate (before you click the mouse) what kind of link they are, and links that provide a pull-down menu of options. Each link specifies the unique URI of its destination node, thus any node anywhere on the web or within a site is reachable.

Unicode

Unicode eliminates the need for using entity references to represent *glyphs* (i.e., characters) whose code points are outside the range of printable ASCII characters. Instead, Unicode provides an unique codespace for each of the world's languages, plus many archaic languages. Each glyph in each language has an unique code point. Glyphs that are common to more than one language (e.g., punctuation) have a single code point that is used by all languages.

Note: The Unicode standard defines the code point for each glyph, not the glyph itself. Separate code points are provided for each diacritical mark, thus characters having diacritical marks can be produced by specifying the code point for the character glyph, followed by the the code point for the diacritical mark glyph.

Different languages can be freely intermixed within the same document. Unicode-compliant fonts are already available which permit the intermixing of as many as 40 different languages with a single font.

For all of the reasons cited above, Unicode will provide a superior solution to the translation of information into different languages.

New Languages

Unicode also provides reserved blocks of codespace for different disciplines to create new language options. Musical, mathematical, and chemical notation languages have already been developed. More special languages for other disciplines will undoubtedly follow. Reserved blocks of codespace are also available for enterprises to create special typograph-

ical symbols (e.g., logos, and other enterprise-unique icons and characters). Users (human and non-human) will be able to treat information in these new languages just like ordinary text, analyzing and manipulating it in any way they see fit.

Resource Description Framework (RDF)

The most potent new feature of XML involves the handling of metadata. RDF is a flexible model for representing named properties and their property values. It allows information about resources to be stored as if it were in a structured database so that it is machine readable. An RDF instance is defined by a named *model* that specifies the syntax and property set for a given type of resource. Complex relationships can exist between resources and properties within an RDF.

Benefits

The benefits of RDF include:

- Much smarter searching
- Greatly improved transfer and pooling of data, such as amalgamation of bibliographies from different enterprises to create global library catalogs
- Greatly improved information management
- Many other novel automation functions are made possible by the machine readability of RDF.

Syntax

RDF uses standard XML encoding as its interchange syntax. The RDF wrapper element marks the boundaries in an XML document between which the content is explicitly intended to be mappable into an RDF data model instance that defines a property set. A `Description` container element child of the RDF wrapper contains the property set.

Suppose, for instance, that an RDF model named `rdf` were defined for this document, and that this model specifies two properties: `Title` and `Author`, both of which are defined in a schema named "s". The complete XML document containing the RDF element would be as follows:

```
<?xml version "1.0"?>
<rdf:RDF>
   xmlns:rdf="URI1"
   xmlns:s="URI2"
   <rdf:Description about="URI3">
      s:Title="FM+SGML Information Design"
      s:Author="Dan Emory"
   </Description>
</RDF>
```

Where:

`xmlns:` is followed by the name of an RDF model syntax or a property schema

`about` is an attribute of the `Description` element that specifies the URI (URI_3) of the document being described

`Title` and `Author` are properties specified (in this example) as attributes of the `Description` container element

URI_1 is the URI containing the RDF model named `rdf`

URI_2 is the URI containing the property schema named s

Observe that the above RDF example is machine-translatable into any of the following English sentences:

> Dan Emory is the author of the resource URI_3, whose title is "FM+SGML Information Design".
>
> OR
>
> Resource URI_3, which is a document entitled "FM+SGML Information Design", was created by Dan Emory.
>
> OR
>
> The document entitled "FM+SGML Information Design", which is identified as resource URI_3, was created by Dan Emory.

If the top-level element for this document has (as it does) attributes named `Title` and `Author`, those attribute values could be machine-extracted, and inserted as the values of the corresponding properties in the RDF. Alternatively, values of the `Title` and `Author` properties of the RDF could be machine-extracted, and inserted into the corresponding attributes of the top-level element for this document. Whichever way it's done, this would assure that the values of properties in the RDF always agree with the corresponding attribute values included in the top-level element of the resource being described.

From the foregoing, it's also evident that RDFs could be produced for MDU_Wrapper and MRU_Wrapper elements within a document. Each such RDF would have an unique URI value in its `about` attribute.

The RDF syntax also accommodates more complex structures than that in the example above to handle, for instance:

- Cases where a single property has multiple values (e.g., two or more authors)
- Cases where there are nested description elements (e.g., the `Author` property could itself be defined as a resource having a nested `Description` element whose `about` attribute identifies the URI for Dan Emory. This `Description` element for the `Author` resource defines two properties: `Name` and `Email`).

Automated Creation and Delivery of Custom Documents from a Database Repository

If all XML-based resources and their RDFs are stored in a searchable database repository, well-formed XML documents, consisting of miscellaneous document fragments (e.g., encapsulated MDUs and/or MRUs, each having its own RDF) could be assembled on-the-fly and delivered to the requesting user. The user originates a database query specifying value(s) in one or more RDF properties. Each time a database hit occurs, the URI in the `about` attribute of that RDF is used to fetch the fragment. The fetched fragments would then be assembled into a document, with the sequence in which the fragments appear being determined by some criterion (e.g., hit rating or parent document).

Automatic Access

Automatic access allows application programs to extract data held between pairs of tags and then manipulate that data automatically for any purpose imaginable. For example, equations or chemical information could be extracted from a document, modified, and then sent to a computer-controlled process of some sort.

Using these automatic access capabilities, many believe that XML could provide superior solutions for applications such as:

- Electronic Data Interchange (EDI), which attempts to define standard ways for companies to exchange orders electronically.
- Electronic Record Keeping in Healthcare, where the ability to pool medical information from many different sources to search for patterns of disease or successful treatments could transform epidemiology.

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com