TechPubTools™

User's Guide

Single Sourcing and API Documentation

tp-tools-ug-02

TechPubTools v. 2.00

Printed in the U.S.A.

Revision History

Revision	Date	Revised By	Revision Summary
02	01/08/2003	Glenn Maxey	Changed information to reflect the way the tools have been modified and their new version level; added several new perl programs, new shell scripts, new filters, new data structures.
01	01/21/2002	Glenn Maxey	Creation of Document; published at beta level.

License and Disclaimer

Permission to use, copy, modify, and distribute this software and its documentation under the terms of the GNU General Public License is hereby granted. No representations are made about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty. See the GNU General Public License (www.gnu.org/copyleft/gpl.html) for more details.

Documents produced by the TechPubTools are derivative works derived from the input used in their production; they are not affected by this license.

Voyant Technologies, Inc. offers these tools and techniques developed for Voyant’s Technical Publications Department in the spirit of professional cooperation among technical writers and in the spirit of open-source tools, upon which part of this solution is based.

However, it should be noted that:

- Voyant is not in the business of creating tools for technical publications; the development of in-house tools in combination with off-the-shelf commercial and open-source tools came about out of necessity rather than as a concerted effort to create a finished consumer product (*which this isn’t*).
- Voyant makes no claims that the HTML system that documents these tools and techniques is entirely representative of what they can do.
  - » Doxygen - an integral part of these tools and techniques - was designed for C/C++.
  - » Doxygen is being used on Perl programs and UNIX shell scripts.
- Voyant makes no claims that any of the Shell scripts, Perl programs, or techniques will work in your environment.
- Voyant makes no claims, expressed or implied that they will:
  - » maintain or support these tools.
  - » incorporate and make available improvements coming from internal or external sources.

You are expected to make any and all modifications to get these to work in your environment.

Notes:

Contents

Notes:

Scope of TechPubTools

This document describes some techniques and tools for *single-sourcing* and application programming interface *(API) documentation* or software development toolkit *(SDK) documentation*.

Single-Sourcing

If your interests are just single-sourcing, what is presented here can assist you in creating an HTML system that is modular and scalable. This assumes that you are using FrameMaker as the source and that Mif2Go or WebWorks Publisher generate mini-HTML systems for your books.

What these tools do is wrap all of these individual systems into one comprehensive system with a table of contents, index, and links to associated PDF files. They make it easier to create modular systems.

API Documentation

If your interests are just API documentation, what is presented here can assist you in creating an HTML system that is modular and scalable. This assumes that you are using code files as the source and that Doxygen or JavaDoc generate mini-HTML systems for your API or SDK projects. What these tools do is wrap all of these individual systems into one comprehensive system with a table of contents, index, and links to associated PDF files.

Extend Me

Although the TechPubTools and techniques were designed specifically for C/C++ API documentation, they can be adopted and modified:

- to handle other programming languages.
- to produce large online documentation systems from multiple FrameMaker books.

Proof of Concept

Experience taught me that modularity should be built into the TechPubTools from the onset in order to save much re-work and re-design later.

Since starting on this project (October 2000), I have employed them and refined them in over 18 separate master projects as of January 2003. Most of the 18 separate projects are still being maintained.

- On the average, each project contains at least three (3) sub-projects from FrameMaker and at least two (2) sub-projects from Doxygen.
- My largest project has two (2) sub-projects from FrameMaker and 38 sub-projects from Doxygen.
- Interestingly, nearly all of the 18 projects contain at least one or two sub-projects that are duplicates, although my source files (and even HTML directories) have not been duplicated. I use symbolic linking and the modularity and flexibility in the tools to publish-it-in-one-place and expose-it-in-many-places.

Although I firmly believe that you can't provide a reader with too much information, I do concede that sometimes too much information in one venue can be overwhelming. Breaking a project into modular components permits re-use of those components in other settings when relevant.

- In one case, I have a comprehensive project with some 40 sub-projects that publishes everything I know about that project.
  - » The comprehensive view is of interest to the owning engineers.
  - » For some of the down-stream internal users, need-to-know issues came up as well as noise in the index and table of contents.
  - » I created separate projects and symbolically linked various sub-project directories in order to generate smaller, focused views of different areas of the project.
- In several cases, individual components of various projects are useful for other related projects.
  - » Certain manuals are needed by both internal and external audiences
  - » Certain internal audiences require different levels of details or components from one or more projects.
  - » I maintain everything about the smaller project in one location. Then I build up larger projects by symbolically linking to component directories of various smaller projects.

Working smarter and not harder.

TechPub Tools Features

The TechPubTools described in this document create an online HTML system with appropriate links to the associated PDF files. The features of the online HTML system include:

- HTML suitable for a website or CD-ROM.
- A modular design with multiple mini-HTML systems coming from various sources.
  - (Mif2Go / WWP) FrameMaker books each can be in their own directory with their own mini-HTML system.
  - (Doxygen / JavaDoc) Code reference components each can be in their own directory with their own mini-HTML system.
- Flexible bi-pane design that can be easily customized; the two panes are the navigation frame and the data frame.
- The navigation frame controls the information that is displayed in the data frame. The navigation frame supports:
  - a comprehensive table of contents with collapsing / expanding levels using an award-winning Java applet. Input to the applet are multiple, generated-from-these-tools script files that are nested and very flexible. The Java applet is available at www.better-homepage.com / java / java-applets-toc.html.
  - a comprehensive index covering all HTML files in the system as well as word-chunking.
  - (DevaSearch) a comprehensive full-text search. This is available at www.devahelp.com.
  - navigation in the index implemented using multiple, generated HTML files and no Javascript.
- The data frame displays the individual HTML topic. The two supported types of HTML topics are “descriptive” topics (books) and “reference” topics (code). Each HTML topic has:
  - a common navigation bar at the top.
  - a second navigation bar that changes depending on whether it is a descriptive topic or a code reference topic.
  - links to the table of contents, index, and its associated PDF file.
- The individual HTML topics as well as the index each have their own template documents. As such, changes to the look-and-feel or navigation can be made in one place and propagated quickly throughout the whole system.

Jump Start

The TechPubTools start where Mif2Go and Doxygen left-off. Alternative products are available for both of these that can get you to the same point.

- Mif2Go is the tool that exports HTML from FrameMaker documents.
- Doxygen is the tool that extracts prototypes and specially flagged comments from the source files.

Assuming that you have figured out how to get HTML output from both Doxygen and Mif2Go, the only additional pieces of information you need are how the configuration / INI files insert the tags that are needed for later processing. These are covered in Chapter 3, "FrameMaker and Mif2Go," and Chapter 4, "Doxygen."

From there and because you're reading this, you probably already unzipped this project and have the template directory structure that you need. It is just a template. Copy it (or unzip the zip file) into a new location that becomes your new build area.

Look into the **00_build_tp_tools.b** UNIX shell script file. You may never have reason to use this script because it uses CVS.

What is important about the **00_build_tp_tools.b** script is that it calls the other scripts in the proper order. Hence, look at what this script is calling and open those script files in sequence to see what they are doing. Copy those scripts and edit as needed to match your directories. Remember to include their call in **00_build_tp_tools.b** for the sake of completeness.

You will need master files (HTML) that reflect what you want to output. You can modify the **voyant_master...html** or make copies of these. (If you copy them to new names, the associated scripts will need to be updated.) These master files need information about your project. This is embedded in HTML comments.

Once you have your source (HTML) in the proper directories, updated master files, and updated scripts, run your scripts.

The output is placed in the **doc_publish** directory. Click on **_start_here.html** to see what you've done.

Chapter 1 Single-Sourcing and API Documentation

The technical writing profession has many buzzwords that want to steer how we produce our documentation. These include:

- single-sourcing
- maintainability
- re-usability
- automation
- web-updatable

The opinions from reputable experts in this field vary about how far we should go in certain directions.

The Naysayer's Argument

Single-sourcing and API documentation generated from code has caused many heated debates in various online forums for technical writers.

Single-Sourcing Naysayers

Some feel that the effort required to implement documentation processes that allow you to write-once and generate-many-outputs isn't cost-effective. They say:

- Implementing the processes and maintaining them distract the technical writers from their primary purpose of writing content.
- Single-sourcing degrades the content of the output formats, because you can no longer apply the text, tone, and information applicable to the different mediums.

- Single-sourcing tools can be crude, can require extensive customizing, can produce imperfect output always needing tweaking, and are thus not always a time-saver.
- Most organizations do not produce enough documentation to warrant such a major investment in a complex documentation system.

API Documentation Naysayers

When the single-sourcing discussion is extended to API documentation, again the opinions vary. If you ignore tutorial, overview, and how-to material and focus only on the reference information, the points of contention are:

- Can source code extraction tools (auto-documentation) generate your reference manual?
- Where should the information for reference manuals reside?
  - In the source code?
  - In documents separate from the source code?
- Who should maintain the reference material?
  - The technical writer?
  - The software engineer?

One side of this argument say that tools cannot get you your API manual. For various company-specific technical and political reasons, they say that the information needs to be maintained separate from the source code by the technical writer.

- Manuals created from auto-documentation tools that extract prototypes and comments from the source code were not as complete or as easy to read as those manuals created "by hand."
- The tools can generate documents that have accurate code prototypes, but don't generate code examples or prolific explanations.
- The source code contains very little information. You get data points, lists of code items and their classification, but little description of what they do, what they mean, or why they are the way they are. Neither the code nor the code comments explain the API to an external user.

- Software engineers shouldn't write the API reference material because:
  - » They don't have the time to step back and think about the code from an API-user's perspective.
  - » They barely have time to comment their code about why something was implemented the way it was.
  - » They can't write, don't like to write, or aren't allowed to write the information in a polished manner.
  - » They might not even have English as their native language.
  - » Whatever they write must be heavily edited.
- Although the software engineer may know (a) what features have been added, (b) why you would use them, and (c) how you would use them, this information rarely resides in the commented code.
- Auto-documentation tools require a level of trust that the software engineers are keeping their code comments up to date.

Reality Check

Whereas all of the points against single-sourcing and against using tools (to produce API reference material) are valid, the counter arguments are more sober because they reflect reality.

- Technical writers are being asked:
  - » (hurdle 1) to produce documents in various output formats. (Single-Sourcing).
  - » (hurdle 2) to update the information in each of those output formats from release to release. (Maintainability).
  - » (hurdle 3) to re-purpose and re-use much of the same information in different manuals and in different output formats. (Re-usability).
  - » (hurdle 4) to streamline the process to improve the accuracy and reduce the margin for human error. (Automation).
  - » (hurdle 5) to keep our customers up-to-date with the latest changes to the documentation. (Web-Updatable).
- Aside from everything else we write, many of us are being asked:
  - » (hurdle 1) to document APIs in (unfamiliar) programming languages.
  - » (hurdle 2) to maintain this API documentation for a fast changing code pool in a development environment with frequent releases.

The rants that follow try to provide some real-world common sense to our plight before plunging into the details of a cost-effective solution that can help us achieve our buzzwords.

- single-sourcing
- maintainability
- re-usability
- automation
- web-updatable

If you ignore the rants (not a bad idea), go directly to the next chapters, and put these open-source techniques to use in your environment, you'll see not only where the nay-sayers above were right and but also where they were wrong.

Single-Sourcing Rant

Many in the technical writing profession have deep-seated views about writing information for the delivery medium. They say that information designed for print is fundamentally different from information designed for online use. Admittedly, a printed manual may not be ideal for online usage.

"Don't let perfect be the enemy of the good."

If you have to weigh "imperfectly" publishing something versus not publishing it at all electronically solely because the text was originally written for print, the deciding factor should be how well your course of action serves your audience.

Withholding information does not serve your audience.

Calls to the Help Desk with questions that an unpublished document could answer do not serve your organization.

Time Heals

Remember that documentation, like software, can be improved from release to release. Over time, we can write or re-write our documentation in such a manner where it will adequately meet the needs of our readers regardless of the medium used to retrieve it.

Or stated another way, any efforts to streamline the text for quick and effective reading online will also benefit a printed version.

Give the Readers Credit

And even if we lack the time to "neutralize" our text to make it acceptable for all media, let's be realistic about our readers. Our readers are intelligent and have good filters. Advertising in newspapers, magazines, television, and the Internet has trained us well.

Analog Not Anal

This means that technical writers can stop worrying about whether traditional terms like *book/manual*, *chapter*, *section*, and *page number* should be purged from the (online) documentation.

1. Our readers are going to read passed those terms anyway to get to the content that is of interest.
2. We shouldn't underestimate the historical significance of those terms and their usefulness in chunking the material and orienting readers.
3. When an online documentation system duplicates (my recommendation) or overlaps (a reader annoyance) with print / PDF, keeping the text identical even down to references to page numbers is a service to the readers. It provides the assurance that the online documentation or PDF documentation is complete and permits using the two in tandem.

4 If the hyperlink works and does indeed take the reader to that information, the "big taboo" of a page number in an online medium isn't going to be a hindrance to usability.

   - Jared Spool of User Interface Engineering states that hyperlinks should give the reader a "scent for the information" and where they will land.
   - If our cross-reference text says "*Turn to page 60 for more information about Topic X*", our readers won't care that there is not a page 60 in HTML as long as the hyperlink works and gets them to the equivalent of page 60 and Topic X.
   - The wording and numbering of cross-references for book ("*The Installation Book*"), chapters ("*Chapter 5 Configuration*") and figures ("*Figure 4.12 Schematic of X*") can and do have a place in print and online help -- much to the chagrin of online help purists.
   - If you must be a purist, become intimately familiar with FrameMaker's conditional text.

Dysfunctional Online

The main advantages of online information are in

- how the information is reached and
- how the information is delivered.

Context-sensitive links from software applications can quickly bring the reader directly to useful, applicable information.

The table of contents, index, and full-text search permits more dynamic access much faster.

When the requirement for printed information becomes less, you can realize an immediate savings on printing and shipping expenses.

Online Disadvantages

The main disadvantages of online information are in the discomfort of reading text on a screen and in the difficulties of printing entire manuals and page ranges.

However, another disadvantage of many implementations of online help is "Lots of Plumbing but Little Water." The implementation has working context-sensitive links from every widget in the interface, yet the resulting pop-up dialog boxes have little content and few links to more information.

Lots of Plumbing but Little Water

A common occurrence when you need assistance with some **Fritz** widget in a dialog box.

- Your first click is on the tiny question mark in the corner of the dialog box just a few pixels away from the minimize and close icons, because there is no obvious **Help** button.
- Your second click on the **Fritz** widget.
- You get a pop-up with the text "*This is where you turn **Fritz** on and off,*" as if the checkbox next to the **Fritz** label didn't already tell you that!
- Your third click makes the next-to-useless pop-up go-away.

In trying to get the online help plumbing to work without errors, the technical writer had little time to provide more information or relevant hyperlinks about what **Fritz** does, what settings interact with **Fritz**, what values are valid, etc. '

Three Strikes and You're Out

A reader is forgiving up to a point. The documentation typically has only three opportunities to prove its value to the reader. When the reader refers to the documentation, searches for information, and doesn't find what they need, their belief after the first two failed attempts is that they are too novice or too stupid with the software to know what to look for; it must be their fault.

However, after the third failed attempt to find useful information in the documentation, the reader quickly surmises that they aren't the ones at fault. The documentation and / or application are at fault. The reader will rarely waste their time again.

Pop-Up Happy Hell

Even when the pop-ups for every widget in the dialog box did contain some information, you had to click three times for every widget in the dialog box to learn everything about that dialog box.

In many cases, a printed manual with all of the information about **Fritz** on one or two pages would have served the reader better than this *pop-up happy* context-sensitive online help.

Context-Sensitive Suggestion

The technical writer was justified in using all context IDs provided by the developer and in avoiding duplication of information (a legitimate maintenance issue). However, they created a usability issue.

In most cases, a given dialog box can usually have all of its **Fritz** widgets described in a single banner topic.

- Look into **ALIAS** and **MAP** sections of the project files to get the individual **Fritz** context IDs to route to your single topic that discusses the entire dialog box. Even in the rare cases when the dialog box coming from software engineering is so complicated that it can't be discussed in one topic, breaking it into two or three topics is still better than letting umpteen individual **Fritz** context IDs dictate that many pop-up topics.
- Look into in-line topics or targets which can get the user to the correct, context-sensitive location of a larger topic.

Note: Jared Spool of User Interface Engineering states from his research and usability studies that "users are not adverse to scrolling if the information is useful and relevant."

Tried and True Printed Manuals

Aside from being portable and having easily understood navigation, printed manuals sometimes have an organizational advantage over online help.

Specifically, many online documentation projects were designed under the mistaken philosophy that the readers will only get to a given topic from a random context-sensitive link directly out of the application. As a result, the technical writer may have neglected:

- To group and organize the topics into a logical sequence.
- To include appropriate entries in the table of contents, which affords the reader on occasion with a much needed overview and scope to the material.
- To implement browse buttons or other means of paging forward and backward "around a topic" for information that would normally be located nearby in a conventional manual, much less paging from cover-to-cover.

Because pop-up topics do not contain non-scrolling regions and other navigational buttons available in "banner" topics, they are excluded often from browsing, the table of contents, and the index for reasons of

look-and-feel and consistency. This can make online documentation even less useful for the "accidental tourist" who relies on discovering new things by paging through what is available.

Note: Context-sensitive help is not the only random way of getting to a topic. Printed manuals have long supported the random nature of the index and table of contents, as well as cross-references to other topics. This is in addition to where a reader might land if they were to flip through the pages stopping at the pretty pictures. Aside from the speed, online documentation really only offer the incremental advantage of getting to a relevant topic directly from the application.

A context-sensitive help system should not be so fundamentally different from a printed manual. The technical writer still needs to:

- develop a logical structure for the information.
- put things in context.
- provide a logical flow.
- write content, because weak topics and thin content aren't as easy to hide in a printed manual that can be skimmed page-by-page.

The Best of Both Worlds

Producing online and printed material need not be a chore where wording and content are tailored to the medium. Single-sourcing not only saves time and effort, but also allows both mediums to improve from the documentation effort. Printed material can benefit from the personal 2nd person language and terse writing style of online material; online material can benefit from the completeness in organization and navigation of the printed material.

The best of both worlds (print and online) can be approached.

Foundation and Structure

Write and organize the documentation with PDF (print) as the initial target even if ultimately intended for online uses (WinHelp, MS-HTML Help, HTML, etc.)

- This forces more thought into document structure, document organization, topic grouping, topic chunking, topic layout, topic wording, etc.
- This can force more thought into the content, because you are primarily concerned about a linear structure rather than the hyperlink plumbing required to make a bells-and-whistle online version.

- A reference chapter that is organized alphabetically is still organized.
- You should plan to provide support for linear browsing that can hit all topics from beginning to end.

Plumbing with Water

Overlay the plumbing for online help (e.g., content IDs, URLs) on the documentation (e.g., markers in the source document) to hyperlink the application to the relevant information in the documentation.

- Plan for **banner topics** (e.g., displayed with a non-scrolling region in WinHelp) rather than **pop-up topics**.
- **Banner topics** allow for better navigation, because navigation buttons (Contents, Index, Browse, etc.) are present.
- As part of the movement for responsible use of **pop-up topics**, encourage the software engineers to have Content-IDs assigned to all widgets in the interface, but have them change the API function call to one that displays **banner topics** instead of **pop-up topics**.
- Map, alias, and route links to the same banner topics or to mid-topic locations as appropriate.
- The technical writer should be making the decision when **pop-up topics** are used where they make the most sense (such as definitions.)

Moving Day

You deliver PDF in addition to your other content-sensitive help. Why?

- If you deliver PDF, you can reduce the number of printed pages that you have to produce and ship. The costs are pushed to the customer, but is a direct savings to your organization.
- PDF files print better than topics from WinHelp or a browser.
- PDF files are laid-out for print and permit printing of entire manuals or ranges of pages.
- With minimal up front planning, each topic in the online help can directly link to its associated PDF file.
- Printed manuals may not always physically remain with the installation machine over time. Online information is often more closely coupled with its associated software application or

hardware in terms of hard-disk location. Hence, online information can remain accessible as long as the application remains installed and working.

Withholding Tax

Don't withhold information. *"Bits are small and CD-ROMs hold lots of them."*

- If your readers don't find the information that they need, who are they going to call? Ghostbusters? Or your expensive customer support department.
- The entire, complete documentation suite can be published and distributed on CD-ROMs. Updates can be made available over the Internet.
- The index/concordance does not have to be limited in its comprehensiveness, because online pages load fast, scroll easy, and can be searched.
- An index is not the same thing as a full-text search. Full-text search is not under the technical writer's control, provides more topic misses than hits, and can be cumbersome to use (e.g., every other click is the browser's **Back** button.)

Note: Paraphrased from Jared Spool and User Interface Engineering:

"The more times the users searched, the less likely they were to find what they wanted.

"The data is quite clear on this: On a single search, users found their content 55% of the time, whereas users who searched twice found their content only 38% of the time. None of the users in our study who searched more than twice ever found their target content...

"Theoretically, as people use the search engine, they should get better at making it perform. After all, each successive interaction is a learning moment -- something that is teaching them the idiosyncrasies of the tool...

"But that's not what we've seen. Either users succeed up front, or things go downhill rapidly."

API Documentation Rant

The beauty of application programming interfaces (APIs) is that once they are defined, they aren't supposed to change. This may give the mistaken impression that its documentation can be written with a one-time push and minimal effort in subsequent releases.

Traditional software documentation projects may require many screen shots of the application whose content the software engineer can re-arrange, rename, and re-locate at any point in time in any release. The documentation needs to keep up in order to be accurate.

API documentation projects, on the other hand, explain "what" the API needs and does, but not necessarily the details on "how" it is accomplished. The "what" isn't really supposed to change or be deleted, otherwise you will break your customer's code which is programmed to the API. The internals of the software below the "what" could completely be gutted and it may only have minor effects to the documentation. Subsequent releases of the documentation mostly only need to be concerned about what was added to the API.

The API code can go through many massive overhauls in the versions leading up to the code freeze of the first release. You can expect:

- The naming of API items to change as the software engineers get in step with legacy code, new conventions, and each other.
- The naming, number, type, and ordering of arguments to API items to change radically as they experiment with what's needed and works and as they conform to others on the project.

The churn in the API software leading up to the release can be even more drastic than the top-level GUI changes which require new screen shots in traditional software documentation projects. In fact, code freeze and beta release to a customer is when promises from software engineering of no more changes affecting the documentation can be believed.

Unfortunately, this is pretty late in the release cycle to expect accurate, complete, and quality documentation, at least for release one.

... Unless the API documentation is revisited from a holistic point of view overlooking the whole organization.

Software Engineering

Software engineering organizations cannot afford to have uncommented code.

- Otherwise the company risks rewriting code every time a new engineer comes on board just because they didn't understand what their predecessor did.
- Software developers know that they are expected to document what they do. Even if their code is never exposed to the outside world, they are expected to comment it.

In more recent years, high-tech has meant high turnover for employees with skills and a thirst for challenge and rewards. Uncommented code is a very big business risk, so it is often mandated in company coding standards and conventions for all software engineers to follow.

On a more personal level, every software engineer should realize that "*if you can't be replaced, you can't be promoted*" and its corollary to software, "*if they don't understand your code, you can't be assigned the whiz-bang new projects and you will be stuck maintaining your old code.*"

Information Repository and Tools

Hence, software engineering organizations are already on the verge of allowing their source code to be the repository for associated reference information. With just a few minor changes to the format of existing code comments, third-party tools can extract that information and create accurate, reliable systems.

Any technical writer charged with maintaining API documentation for any quickly changing code base will one way or another tell you that the task shouldn't be done with tools.

The extent of the tools depend on many factors. At the very least, the tools need to flag differences in API syntax from one build to the next so that finding which parameters change isn't a manual job of digging through the code to visually match a code item with the external documentation.

Reusability

Consider one of the buzzwords of software development: *code reusability*. In order to employ existing code in other applications, the engineer has to know what the code does. More often than not, the engineer turns towards the header and source files to get all of the information they need directly from the code.

Based on where they obtain information and the requirement to comment their code, the software engineers are not very far away from more robust code extraction tools. What they may lack are:

- A template for code comments.
- Code comments having the proper format for an extraction tool.
- Time and desire to mark-up and/or alter existing files.

Send in the Tech Writer

This is where a diligent technical writer comes into play. The technical writer can:

- Help define the minimum template for all code comments.
- Insert templates into the code.
- Convert any existing code comments into the new format and fix any language errors in the process.

If the technical writer can't figure something out, they can flag it for the software engineer, who ultimately still has the responsibility to comment the code.

Whereas technical writers may also have limited time and desire, such work is in more in line with their job description. A tool that extracts information from the source-code becomes self-serving for the technical writer.

The extraction tool working on the source code:

- Exposes the deliverables of the software developer in an organized fashion.
- Gives the technical writer 80% or more of the reference material for the API documentation.
- Improves the accuracy of the documentation suite, because prototypes come directly from the code.
- Improves communication between technical writing and engineering.
- Shifts the effort of documenting and maintaining the API reference material to the owning engineering.

What You Get

This work directly benefits the technical writer in all subsequent releases, because the software engineers will start maintaining their reference material and reusing your templates for new code items.

The technical writer's efforts also provides some good public relations with software engineering if re-formatting and re-writing of code comments does not fall in bulk onto the software engineer's to-do list.

Let's not forget the benefits to the whole organization through:

- Improving the readability of the code.
- Helping bring new employees up to speed.
- Enforcing coding practices.
- Providing a basis for code review.
- Easing the burden of the technical publications group.
- Providing a comprehensive HTML system to object-code and source-code customers.

Source Code Extraction Tools

In general, tools that extract things from the code, sometimes called auto-documentation tools, are written by software developers for software developers. This means that the tools have already made most of the important decisions about what should be documented. And they can do their work automatically.

Although Doxygen (www.doxygen.org) is mentioned as an open-source solution to create reference documentation directly from the code, other tools are available.

I've run Doxygen on code that had no comments that Doxygen recognized. The HTML output (with no plain English) was still pretty impressive and better than I could have done by hand. Doxygen:

- Traces the code, something that I wouldn't want to have to do by hand.
- Documents who is using what function, where that function resides, etc.
- Generates hierarchy diagrams.
- Hyperlinks to code items that it knows about when it comes across it other areas.
- Has optional features to show source code and also to hyperlink you to specific areas in the source code.

The framework that it builds should not be underestimated and would be really hard to create by hand in a (FrameMaker) manual.

Concerns about Auto-Documentation Tools

Manuals created from auto-documentation tools may not always be as complete or as easy to read as those manuals created "by hand." The counter arguments to this are:

- If your organization presently has no API documentation, source code extraction tools give you a significant head start.
- Even those wonderful, easy-to-read, manually created API documents did not evolve overnight into completeness. If your tool has holes in its coverage:
  - » You manually document the holes in the system and let the tool help do everything else. Don't re-invent the wheel.
  - » You can have your software development organization enhance the open-source tools to achieve what you need.
- Completeness is relative.
  - » Many times the code item name and its syntax are descriptive enough to the audience when viewed from a header or source file.
  - » If the code items that the audience is expected to use are well documented, the documentation may not need to be as complete or as detailed for other code items that might be exposed.
  - » The API evolves and so does the documentation. Do what you can when you can, and let feedback from others (internal and external to your company) help prioritize what needs to be improved.
  - » The completeness of the API documentation can be a recursive process.
- With the appropriate company processes in place, the same excellent writing can be achieved regardless of where the information resides.

Writing Quality in the API Documentation

The issue of writing quality in the API documentation may never be resolved, because the software engineers own the code. The trade-off for getting them to update their comments to reflect the code at the same time they are making changes to the code is probably worth any language errors that they might introduce and that don't get fixed immediately.

If the software engineers know their writing (e.g., comments that get extracted into the API documentation) will be exposed to a wider audience:

1. They can willingly let the technical writer edit their comments.
2. They can let management dictate that the technical writer edit their comments.
3. The technical writer can leave it as the engineer wrote it.

With regards to number 3: let's look at the big picture. Is it really going to matter to the API documentation audience - *other engineers* - whether the engineer who wrote the comments had a typo, a run-on sentence, or other language blunders? Will the audience - *other engineers* - even see the error *[maybe]* or care *[no, not if the other information is accurate]*? The audience cares more about information than its delivery. They also recognize that they can't throw stones at things that their own code might be guilty of.

Note: This is not meant to advocate releasing API documentation that is riddled with errors in the form of typos and grammar mistakes. It is meant to temper our inner control freak who might impose unrealistic expectations and processes on others in our effort to obtain grammar-error-free documentation nirvana. With compressed development cycles and frequent releases, there will always be next release to incrementally improve things more. When issues of wording ownership arise in source files, next release can often be a better time to divorce text from its author (the software engineer) with less resistance.

Auto-Documentation Benefits

One of the benefits of auto-documentation tools is that reference information is located where it belongs: in the source code. It is not separated from the code in a FrameMaker book that technical writers must constantly maintain through every whim of the software engineer's ordering, adding, and deleting of arguments. Those things are taken directly from the declarations and definitions. It's automatic, and code references in the published documentation are never out of sync with the code, providing that the tool is run against the code at code freeze or other applicable points in time.

As was mentioned, auto-documentation tools can generate documents that have accurate code prototypes, but don't generate code examples or prolific explanations. However, the theory is that if the technical writer isn't spending their time keeping the syntax up to date, they will have more time to generate the code examples or prolific explanations, which they would have to do anyway.

The Code Can Contain More Information

Another argument against auto-documentation tools was that the source code contains little information. The source code contains data points, lists of code items and their classification, but little description of what they do, what they mean, or why they are the way they are. Neither the code nor the code comments explain the API to an external user.

Just because the source code doesn't contain more information, doesn't mean that it can't.

And even if the source code never does contain more information due to company politics, this does not mean that auto-documentation tools can't still be employed. Many such tools provide hooks that allow external documentation to be included in the build, in addition to hyperlinking to and from an external systems. Auto-documentation tools can still provide many benefits to the technical writer even in the rare cases of resistance from the software engineers.

Software Engineers as Technical Writers

Just because software engineers may have a list of deficiencies that make them less than ideal for writing API reference material, this should not be used to dictate where the information resides. They are the subject matter experts. Maintaining information in the source files puts it "in their face" in the hopes that they do update the information as soon as it changes. The process and the working relationship with the technical writer will determine the final quality regardless of where the information physically resides.

Auto-documentation tools require a level of trust that the software engineers are keeping their code comments up to date. If they can't keep their own code comments up to date, what makes us think that they'll remember to tell a technical writer?

The Future of (API) Documentation

The immediate future for software is that the source code has to be able to describe itself. The code comments need to be in a standard format that can easily be extracted with the syntax into online reference systems. This is paramount for both open-source and proprietary code if software companies want any longevity and reusability out of their code.

Whereas I believe that XML will (initially) provide only marginal added-value to run-of-the-mill documentation, this is not true when it comes to API documentation. (a) If the source code contains comment that can be extracted, (b) if those comments can contain HTML syntax (as is the case with open-source Doxygen comments), and (c) if the need remains for the source code to describe itself, then XML tags within the code comments is a natural extension.

If the source code itself is well commented and marked up appropriately with descriptive XML tags, the hooks are then in place for having the source code for software solutions served up from data bases.

Code Generation

When the ever-increasing power of processors is taken into consideration, the speed-related drawbacks to interpreted languages (like Perl) become less noticeable. The fact that you can view the source before you execute it (when not compiled) becomes a significant advantage for security, for reliability, and for custom solutions.

Combine this with database tools that use XML tags and self-documented code.

What we'd have would be source code that is truly re-usable, because tools would be able to search large databases (CVS repositories) and find useful, commented code fragments.

We would then be on the verge of having Code Generation tools that can find, combine, and compile any number of permutations of distributed source code.

Software Public Libraries

If I could start a business that would make me wealthy, topple Microsoft, and change the world for the better, I would create a **`Software Public Library`** (SPL).

The SPL has two initial revenue streams:

- Code Check-In.
- Code Check-Out

Each would have sliding scale depending upon the expectations placed upon SPL with regards to test, verification, certification, compilation, and packaging. In other words, if you want to check out a

fully certified and packaged executable, the cost will be higher than untested code submitted by some lone Software Engineer on a pig farm in Sweden and that you have to build yourself.

Get More Thinkers Involved

Today the price of a finished commercial program (executable) is less than the price of acquiring its source code. The new paradyn keeps the executables priced affordable (at or below present levels) while making source code significantly cheaper. Although the sliding scale still accounts for levels of testing and certification, acquiring the source code should be a realistic and affordable proposition to any registered, independent software engineer in the world. The intent is to get more minds working at improving the software that the world uses.

The software engineer is encouraged to check in their improvements to the SPL. The nature of their fix and its urgency can determine how quickly the normal SPL processes move the change from untested to certified. Of course, larger software organizations can pay to speed up the process if they don't become an SPL-certified contributor who perform the tasks required to make a release available in the SPL at all levels.

Royalties for Their Efforts

A rating system based on the number of lines of code, a weighting of the functionality in the program, and other factors can determine a royalty percentage that can be paid to the software engineer for that fix that is used in all subsequent for-profit derivative programs and certified executables.

(New) Copyright Protection

For brand-new technology, the SPL could

- Limit the source code exposure for a short time depending upon the level of copyright protection.
- Limit check-in and availability of altered source code unless the new implementation is truly unique and better.
- Negotiate appropriate fees and royalties for derivative products to include the technology.
- Facilitate exposing the technology's interface so that others can build around it.

Because proportional royalties are paid for derivative products, no revenue is lost for contributions that remain in use. The incentive for proprietary solutions becomes less. The life of the solution - in whole or in part - and hence the length of time that royalties continue to be paid becomes directly proportional to the openness of the solution that fosters wide-spread acceptance. If the solution works and is open, and if nobody is being robbed for their creative efforts, the incentive to "re-invent the wheel" decreases while the incentive "to spend our time building on what already works" increases.

To bring the copyright policies of the world in line with one another, the SPL is intended from the onset to be a government regulated monopoly.

- On the one hand, all published / commercial (software) works would be required (by law) to be made available in the library: executables, DLLs, object, and commented source code.
- On the other hand, check-out and download requires registration and payment. The check-in process can determine the extend to which a new product is unique or derivative. Another sliding scale determines the percentages and duration for royalties back to the originator.

New SPL Revenue Streams

The "archeology" division of SPL would actively solicit source code for outdated, unsupported software (like DOS, WordPerfect 5.0, BIOS, etc.) After commenting and tagging, it could be made available.

The "custom solutions" division of SPL would handle requests for enhancements and porting to old and new platforms and operating systems. SPL could help manage supply and demand in terms of contracting and scheduling software engineers based upon requests.

The "test and certification" division would provide various levels of quality assurance, virus detection, verification, certification, and integration.

The "code integration and generation" division of SPL would automate the process for individuals to specify an application made out of components.

What You Get

In the future,

- Your hardware can be supported much longer.

- Your software reliably does what you need it to do.
- You can afford custom software.
- Your data remains compatible.
- You can do it yourself if you're so inclined.

The keys to this better and brighter future for software are knowing:

- What code is available [all versions].
- What is in the code [XML tagging and extraction].
- What the code does [API documentation and code comments].

Chapter 2 Environment and Tools

Aside from discussing the environment that I work in and the tools that I acquired to achieve my organization's goals, this chapter also serves as a disclaimer regarding the suitability of TechPubTools in your environment.

Your network environment can vary greatly from mine.

What is important to note about my environment is that I use a UNIX file server that both my Windows 2000 and UNIX workstation access. I use an X-Windows emulation program on Windows 2000, so that I can go back and forth between my Windows applications and my UNIX applications without thinking about it.

There are many ways to achieve the same or equivalent functionality that I enjoy. Moreover, it is possible to achieve the same goals of the TechPubTools in a pure-Windows environment and possibly even a pure UNIX/Linux environment. This is left as an exercise for you to accomplish.

My Environment

Below are the operating systems and off-the-shelf tools that I use or have access to in order to implement the TechPubTools.

Microsoft *Windows 2000*, version 5.0095 with Service Pack 1. This is my main work environment from which I operate FrameMaker, Mif2Go, my HTML editor of choice, and DevaSearch. For more information, refer to www.microsoft.com.

Adobe's *FrameMaker 6* running under Windows 2000 is used to create the printed and PDF documentation. FrameMaker (FM) files are single-sourced into an HTML system. For more information, refer to www.adobe.com.

FrameMaker is used to create “structured” information that is intended to be handled like a printed book even if electronic in nature. In particular, this means that:

- Information is organized with a table of contents.
- Die-hard readers can go from the beginning to the end of this documentation and not miss any topics.
- All topics link to their previous and next topics, which helps orient the reader when displayed from the index or table of contents.

Omni Systems Inc. *Mif2Go*, version upg33u20.zip, an add-on program for FM that among other things can convert from FrameMaker’s MIF format to HTML. For more information, refer to www.omsys.com.

- WebWorks Publisher Professional (WWP) could also be used to achieve the same goal as MIF2Go.
- At the time of writing, Mif2Go was a lower cost solution over WWP and seemed to provide more flexibility.

Ulli Meybohm’s *HTML EDITOR Phase 5* (Release 22.09.1999). This is a FreeWare program in German for editing HTML files. If you already have a good HTML editor or don’t understand German, you don’t need this. For more information, refer to www.meybohm.de.

Sun Microsystem’s *Solaris 2.7* (UNIX) on a Sun workstation. This is a machine owned by Voyant’s Software Engineering department that I use infrequently to run Shell scripts and Perl programs. More importantly, we use a UNIX file server that both Windows and UNIX can access. This is not needed if you port everything into a Windows-only environment. Refer to www.sun.com.

- UNIX Shell scripts are used to create control much of the process, because this is Voyant’s development environment.
- Some of the Shell scripts perform CVS commands, a source code management and version control system.
- Batch files in the Windows environment could accomplish much of the same thing.

StarNet Communications Corp.’s *X-Win32*, version 5.1.1. This is an X-Windows desktop emulation environment for Windows 2000. This is what gets me access to the Sun workstation. This is not needed if you port everything into a Windows-only environment. You can find many ways to access a UNIX environment, if that is even a requirement. For more information, refer to support@starnet.com.

Dimitri van Heesch's *Doxygen*, version-1.2.11. This is under the terms of the GNU General Public License (e.g., open-source). This can be ported to a Windows-only environment. For more information, refer to www.doxygen.org.

- Doxygen is a documentation system for C++, Java, IDL (Corba, Microsoft and KDE-DCOP flavors) and C. Doxygen is an open-source tool used to extract code prototypes and specially flagged code comments from C/C++ source code.
- JavaDoc is another open-source tool that accomplish many of the same things for the Java language; JavaDoc output could easily be incorporated into this solution.
- Doxygen is also used imperfectly on Perl files and on an in-house Pascal-like programming language (IVE) files. This required the use of input filters which fake out Doxygen in what it sees and recognizes.

DevaHelp's *DevaSearch*. This is used to implement the full-text search. Although the tool is used in the Windows 2000 environment, the resulting HTML and Javascript is cross-browser, cross-platform. For more information, refer to www.devahelp.com.

Perl, the programming language used in the scripts. Perl programs are used to change the content of the HTML files in specific areas. You'll need the Perl interpreter for your environment. I use Perl in the UNIX environment. Perl is available in Windows-only environment. For more information, refer to www.perl.org.

Just because this solution uses these off-the-shelf tools does not mean that other tools cannot be used instead or in addition. This open-source solution is meant to provide you with ideas and foundation upon which your technical publication's solution can be built.

Home-Grown Tools

The following are tools that were developed in Perl at Voyant Technologies, Inc. and are the heart of these TechPubTools. They are listed with a brief description and are covered in more detail in later chapters as well as the code reference portion of the HTML system.

voyant_nav.pl swaps out the head information, navigation controls, and copyright information. It also extracts temporary files used for the table of contents and index.

voyant_mt_app.pl uses the temporary files for the table of contents and generates a series of script files for the Java applet that are the master navigation over the whole system.

voyant_indexer.pl uses the temporary files for the index and generates a series of HTML files that are the master index over the whole system. It also performs word-chunking that expands the number of index entries and turns it into more of a concordance, a useful feature for programs.

voyant_latex.pl changes template LaTex files that were generated by Doxygen before LaTex is used to create PDF files.

dox_bug_filter.pl performs input filtering of source files into Doxygen to change elements, such as comment styles for more reliable output.

dox_ive_filter.pl performs input filtering of source files into Doxygen. The input files are known to be in the IVE language (Pascal-like). This imperfect filter makes those input files look more C-like in order for proper handling by Doxygen.

dox_comment_chg.pl performs input filtering of source files into Doxygen to change comment styles for more reliable output.

dox_chg_not.pl changes Doxygen comment styles. Intended to be used on source files that were initially documented using a style that the coding conventions later changed.

pl_comment_chg.pl performs very imperfect input filtering of source files into Doxygen. The input files are known to be in the Perl language. This imperfect filter makes those input files look more C-like in order for proper handling by Doxygen.

Although these tools are implemented and run in the UNIX environment, they could be used in the Windows environment and called from a batch file. (There are only a few areas where explicit UNIX system calls are made to generate lists and copy files. Equivalent system calls to DOS commands can accomplish the same task.)

Directory Structure

tp_tools The top-level directory used in the creation of the online HTML system and contains the UNIX Shell script files (given with "**b**" extensions) and the master template files for the navigation, table of contents, and index. It also contains the Doxygen project files and other files used in the generation of HTML.

common_files Contains GIF and CSS files that are used in all subdirectories of doc_publish. Also contains default HTML files that are part of the code reference sections. Depending upon the content of the source code, Doxygen overwrites these files.

doc_publish Contains all files in the HTML system intended for distribution to readers. Moreover, everything from this level down is either under source control (CVS) or can be re-generated. This level contains the series of HTML files that make up the table of contents and index, as well as the introductory topic and PDF overview.

book_help_on_help Contains all HTML and GIF files associated with the how to use the help system. The source FM files are located elsewhere. This directory also contains copies of the Mif2Go INI files that configured the HTML output generation.

book_tp_tools Contains all HTML and GIF files associated with the description of TechPubTools. The source FM files are located elsewhere (in **src_fm**). This directory also contains copies of the Mif2Go INI files that configured the HTML output generation.

book_... Other directories can be created with information that is maintained in FrameMaker and exported to HTML using a tool. In order to support additional book directories, the master files need to be updated accordingly as well as the controlling shell scripts.

cref_tp_tools Contains all HTML and GIF files associated with the API source code being documented. The files are generated using Doxygen or another tool. The source code files are located elsewhere.

cref_... Other directories can be created for information that is extracted from the source code into HTML using a tool. In order to support additional cref directories, the master files need to be updated accordingly as well as the controlling shell scripts. When using Doxygen, the respective Doxygen project files need to be created which generate their output here.

DevaSearch Contains all HTML and JS files associated with the full text search. The files in this directory are generated after all other aspects of the system have been created.

print_pdf Contains all PDF files associated with the other directories. The PDF files could reside elsewhere, but a design decision was made to collect them all here for the user.

src_fm A directory containing the FrameMaker documents used in generating an HTML system. In the case of this TechPubTools project, this is the source directory the Mif2Go INI files that then generates HTML in **book_tp_tools**.

src_perl (Actually a symbolic link to) a directory containing the Perl code used in generating an HTML system. Refer to the **perl** directory. This was included here to simplify the creation of the zip file.

zdoc_merge A temporary directory used in the generation of the series of HTML files for the index and table of contents. The temporary directory provides more control over what is included and excluded and facilitates testing.

zlx_tp_tools A temporary directory used by Doxygen to hold the LaTex files one API source code project and to generate associated PDF files.

zlx_... Other temporary directories can be created for API source code. They are used by Doxygen to hold the LaTex files and to generate associated PDF files.

perl A directory containing the Perl code used in generating an HTML system. In the case of this TechPubTools project, this is also the source code directory for the **tp_tools.dox** Doxygen project. Doxygen generates HTML in **cref_tp_tools** and LaTex/PDF in **zlx_tp_tools**.

Chapter 3 FrameMaker and Mif2Go

Adobe's *FrameMaker* is used to create the printed and PDF documentation. FrameMaker (FM) files are single-sourced into an HTML system using *Mif2Go*.

It is beyond the scope of this manual to discuss all of the details for effective use of either of these tools.

- For more information on FrameMaker, refer to www.adobe.com.
- For more information on Mif2Go, refer to www.omsys.com.

What is appropriate for this chapter are the tips-and-tricks that are employed to get these tools working together with each other and the TechPubTools to achieve a comprehensive HTML system.

FrameMaker

FrameMaker (FM) is used to create "structured" information that is intended to be handled like a printed book even if electronic in nature. In particular, this means that:

- Information is organized with a table of contents (TOC).
- Die-hard readers can go from the beginning to the end of this documentation and not miss any topics.
- More realistically, it means that when readers land on a topic by whatever means (e.g., TOC, index, external hyperlink, context-sensitive links), the book's structure allows them to easily acquire background information from previous topics and more detailed information from the next topics.

However, just because a FM book structures the information, this does not mean that it any less useful in an online format. In fact, I believe that this effort to provide organization and order makes the

information even more useful when presented online, mostly because it provides the reader with at least one form of navigation with which they are already familiar.

FM Formats

One could argue that the key to effective use of FrameMaker is:

- to have a bullet-resistant style/format guide.
- to adhere to the formats defined.
- to minimize format overrides.

When single-sourcing the FM files into other output formats, such as HTML, the enforced use of the style guide plays an even greater role. Everything accomplished in FM needs to be mapped in some manner to an appropriate construct in the output medium. Moreover, online output imposes requirements for both consistency and flexibility on the styles and formats.

Generally, the formats employed in a FrameMaker document tailor it for print or PDF purposes. When exporting the document into, say, HTML, the export tool needs to be told what the topic boundaries are.

Our style guide has:

- Five (5) paragraph formats that define the chapter chunks:
  - **TitleChapter**
  - **TitleAppendix**
  - **TitleFront**
  - **TitleIndex**
  - **TitleIntro**
- Three (3) paragraph formats that define the section chunks:
  - **Head1**
  - **Head2**
  - **Head3**

All eight (8) of these paragraph formats are used in the generation of the Table of Contents. Moreover, they are used to define sections that become stand-alone HTML files when exported.

In other words, these FrameMaker paragraph formats can be followed by any number of any other paragraph format. Only when another of these eight (8) formats appears is a new HTML document created. The text used with these eight formats help determine the HTML file name and subsequent hyperlinks.

On occasion, these **Title** or **Head** formats are followed by only a single sentence or paragraph, which is acceptable in print / PDF but results in a very small HTML topic. To address this, three additional "*keep with previous*" (**Kwp**) paragraph formats are defined.

- **Head1Kwp (=Head1)**
- **Head2Kwp (=Head2)**
- **Head3Kwp (=Head3)**

These **HeadnKwp** have the exact same definitions as their associated **Head** format and are used in the creation of the Table of Contents. The only difference is that these **HeadnKwp** formats are purposely missing from any definitions used by the HTML export tool for HTML file splitting. When a **HeadnKwp** is used in place of the **Headn**, it does not cause the forking of a new HTML file.

The intend of these additional (duplicate) paragraphs is to give the technical writer more control over the HTML file size and topic chunking. When the technical writer has nearly completed the manual for print / PDF generation, they can go through the FM book and determine which sections are small and could be combined with the next or previous section. They change the **Headn** format at the boundary between two sections to its associated **HeadnKwp**.

Note: Your format names may be different than those that Voyant uses. The point of this discussion is tip on how the HTML export tool can be told not to generate new HTML files.

Mapping FM Formats to HTML Constructs

HTML purposely has just a few constructs with which to tag information.

- Headings: **<h1></h1>, <h2></h2>, <h3></h3>**
- Paragraph text: ** **
- Pre-formatted text: **<pre></pre>**

For proper viewing in HTML, all FM formats need to mapped to one of these. Thankfully, the cascading style sheet (CSS) permits the definition of classes that can further define formatting to one of these tags.

Even though my FrameMaker environment has four levels of paragraph formats - **Title<...>, Head<1>[Kwp], Head<1>[Kwp], Head<1>[Kwp]** - they are all mapped to **h1**. The **<h*n*>** tags in HTML are used for display and not necessarily hierarchical relationships.

```
[ParaStyles]
TitleChapter=h1
TitleAppendix=h1
TitleFront=h1
TitleIndex=h1
TitleIntro=h1
Head1=h1
Head2=h1
Head3=h1
Head1Kwp=h1
Head2Kwp=h1
Head3Kwp=h1
```

The Voyant documentation sometimes has programming code or command-line syntax that would be incorrect if permitted to automatically word-wrap based on the browser's Window size. The appropriate FrameMaker **code<...>** paragraph format provides the correct font and layout characteristics in the print / PDF. In order to assure that no inadvertent word-wrapping occurs in the HTML output, these FM paragraph formats are mapped to the HTML **<pre>** tags.

```
[ParaStyles]
Code=PRE
Code1=PRE
Code10pt=PRE
Code11pt=PRE
Code2=PRE
Code3=PRE
Code4=PRE
Code4 (body Indent)=PRE
Code5=PRE
Code9pt=PRE
CodeAPI=PRE
```

It is assumed that all other FrameMaker styles encountered during the export to HTML will be mapped to a **<p>** style with an appropriate class refinement (that the cascading style sheet controls).

The mapping of styles using **[ParaStyles]** is a construct that is part of Mif2Go in one of its INI files. However, the ability to map styles is a fundamental part of any other Help Authoring Tools, such as WebWorks Publisher (WWP). Such tools will have a configuration tool or file that can establish the same settings.

Online Use and Conditional Text

If you take the position of many in the technical writing profession that the information should be tailored for the delivery medium, then FrameMaker is the tool that can help achieve this goal while keeping with the single-source techniques.

Using FrameMaker's conditional text feature, a single FM document can contain language that is specific to the output format. This is most likely to occur when creating cross-references. If your text isn't completely neutralized from:

> (Print / PDF) Please refer to "*Subject X*" on *page n*.
>
> (Online) Please refer to the topic "*Subject X*."

to simply:

> Please refer to "*Subject X*".

you can still employ conditional text so that both the print and online versions co-exist in the same document.

FrameMaker offers much freedom in how you structure your conditional text. However, most who use conditional text recommend that it be employed at the paragraph level, rather than at the sentence, phrase, or word level.

The issues with conditional text fragments smaller than paragraphs center around where spaces and punctuation should appear. Who owns the spaces and punctuation? The parent text or the conditional fragment.

Particularly when multiple writers are involved, it easy to create inadvertently fragments that don't have enough or have too much leading / trailing spaces and punctuation for the selected condition.

Cascading Style Sheet

The cascading style sheet (CSS) in HTML can help achieve both consistency and flexibility in the HTML output. However, a prerequisite for the CSS is that the source FrameMaker material was consistent in its use of paragraph formats.

The CSS centralizes definitions for the display and layout. Flexibility is obtained by making some definitions less specific, such as a font-family like "serif" or "sans-serif", which makes more allowances for what the fonts the reader has available.

In our case, the Mif2Go product does an efficient job of exporting the FM formats into the CSS and can change the formats as it converts.

Mif2Go

Omni Systems Inc. *Mif2Go*, is an add-on program for FrameMaker that among other things can convert from FrameMaker's MIF format to HTML.

Mif2Go has employs INI files to establish the configuration. It is beyond the scope of this document to discuss all Mif2Go options and their configuration in the INI files.

Instead this section will focus on the aspects of the configuration that allows the HTML output from Mif2Go to be plugged into a bigger HTML system.

Note: This section discusses items that are specific to Mif2Go. However, it should be noted that WebWorks Publisher Professional (WWP) could be used to achieve the same tasks as Mif2Go. At the time of writing, Mif2Go was a less expensive than WWP.

The following directories contain examples of both the `mif2htm.ini` and the `mif2htm.ini` files.

- `doc_publish\book_tp_tools`
- `doc_publish\book_help_on_help`

Fonts Mapping

The Mif2Go product does an efficient job of exporting the FrameMaker paragraph and character formats. It can change the formats as it converts.

For example, the serif fonts (such as Time New Roman or Palatino) have historic readability reasons that make them appropriate for body text intended for print. The serifs are the little horizontal marks at the tops and bottoms of letters. Over time and practice reading, the human eye is trained to look at the shapes of words rather than the individual letters within the word. The serifs provide more definitive shapes to words to facilitate their recognition.

However, monitor resolution of the serif's as pixels can seriously degrade the readability when used online. The "chunky" nature of pixels makes the serifs more of a hindrance for recognizing word

shapes. For this reason, sans-serif fonts (such as Ariel or Verdana) are often used more often for body text online. Verdana is the online font presently recommended by Microsoft for online documentation.

Aside from the font definitions inside of FrameMaker's paragraph and character formats, fonts are defined in:

- The `mif2htm.ini` file.
- The cascading style sheet.

The main font items that the `mif2htm.ini` file defines are re-mapping of print fonts to online fonts.

```
[Fonts]
Palatino=Verdana
GillSans=Times New Roman
AvantGarde=Verdana
GillSans Light=Times New Roman
Times=Times New Roman
Helvetica=Arial
Courier=Courier New
HelveticaNarrow=Arial Narrow
Helvetica-Narrow=Arial Narrow
Century Schoolbook=NewCenturySchlbk
Common Bullets=CommonBullets
```

The Voyant corporate style guide for printed documentation recommends:

- GillSans (sans-serif) for titles and headings.
- Palatino (serif) for body text.

However, online user's of Voyant's documentation cannot be expected to have GillSans or Palatino. Moreover, Voyant's use of serif and sans-serifs fonts could be troubling when read from a monitor. The `mif2htm.ini` file maps (serif) Palatino to the sans-serif Verdana font and the (sans-serif) GillsSans to the serif Times New Roman font.

In addition, Mif2Go is used to create the first cascading style sheet (CSS). Mif2Go creates classes for all paragraph formats in the CSS giving these the same properties as the owning FrameMaker document unless mapped otherwise in the `mif2htm.ini` file.

The CSS is further modified by hand to achieve our desired online look-and-feel. An important change the we implement is the use of **`font-family`**. This means that our CSS definitions specify one or two fonts that their system probably has, like Verdana and Ariel. However, if those aren't on the system, we specify "serif" or "sans-serif" and let the browser application locate an appropriate font.

File Splitting

The `mif2htm.ini` file specifies how to chunk the information coming from FrameMaker into individual HTML files. The Voyant style guide for FrameMaker has:

- Five (5) paragraph formats that define the chapter chunks:
  - **`TitleChapter`**
  - **`TitleAppendix`**
  - **`TitleFront`**
  - **`TitleIndex`**
  - **`TitleIntro`**
- Three (3) paragraph formats that define the section chunks:
  - **`Head1`**
  - **`Head2`**
  - **`Head3`**

All eight (8) of these paragraph formats are used in the `mif2htm.ini` file in the **`[HtmlStyle]`** section to define the exported stand-alone HTML files.

```
[HtmlStyles]
TitleChapter=Split Title Filename
TitleAppendix=Split Title Filename
TitleFront=Split Title Filename
TitleIndex=Split Title Filename
TitleIntro=Split Title Filename
Head1=Split Title Filename
Head2=Split Title Filename
Head3=Split Title Filename
```

`Split` - specifies that the paragraph format to the left of the equals (=) is to be used to start a new HTML file.

`Title` - specifies that the contents of the paragraph format is to be used in the **`<title></title>`** tags of the HTML file.

`Filename` - specifies that the contents of the paragraph format is to be used as part of the HTML filename. Mif2Go removes whitespace and punctuation in the name.

Note: If you do not supply the Filename parameter in the **`[HtmlStyle]`** *section for a given FrameMaker format, Mif2Go creates a name from tags that are internal to FrameMaker. This is the method that the makers of Mif2Go recommend, because FrameMaker assures that these tags are unique.*

The advantage of using the **`Filename`** parameter in the **`[HtmlStyle]`** section is that your HTML filenames are more meaningful.

The disadvantage of using the **Filename** parameter in the **[HtmlStyle]** section is that you have to assure that the titles of your paragraph formats (e.g., **Title<...>, Headn**) are unique within your chapter (or book). If they are not unique, the HTML files get overwritten by any subsequent paragraph formats in the chapter (or book) with the same name that Mif2Go generates.

As will be discussed later, Mif2Go supports chapter **INI** files. These can be used to append a prefix or suffix to the generated HTML filenames. Hence, generated HTML filenames coming from different chapters do not conflict, although they do have to be unique within a chapter. Whereas it is admittedly more difficult to assure uniqueness of titles throughout a book, reducing the title uniqueness requirement to within the chapter is not as much of a burden on the technical writer.

For other technical reasons (ordering of the table of contents), chapter **INI** files are employed to assign two-digit prefixes that provide a clue as to which chapter the HTML topic belongs and at the same time resolve overlap in potential HTML filenames from different chapters. A further advantage is that, together with the **Filename** parameter, the readers (and technical writers) can orient themselves within the documentation just from the HTML filename.

Post-Processing Tags in Splitting

The `mif2htm.ini` file defines where the split for new topics happens using the **[HtmlStyle]** section. In addition, Mif2Go supplies pre-defined macros that can be specified in the **[Inserts]** section of the `mif2htm.ini` file.

SplitHead - adds information to the generated HTML file in the **<head>**. Among other things, this is used to specify variables, such as the previous and next topics, so that they remain available within the topic.

SplitTop - adds information to the generated HTML file at the top of the **<body>**. This is used to define navigation control.

SplitBottom - adds information to the generated HTML file at the bottom of the **<body>**. This is used to define copyright information.

The TechPubTools takes advantage of these Mif2Go macros to facilitate later post-processing by the voyant_nav.pl tool.

Specifically, the TechPubTools employ standard HTML comments that reside in the file but are not displayed by the browser. The comments are typically used as pairs (**begin** and **end**) and contain a tag name that TechPubTools recognize.

<!-- begin voy_common_top --><!-- end voy_common_top --> used for system-wide common navigation.

<!-- begin voy_fm_book --><!-- end voy_fm_book --> used for navigation that is specific to a FrameMaker book. In particular, this means buttons or hyperlinks for previous and next topics.

<!-- begin voy_dox --><!-- end voy_dox --> used for navigation that is specific to the Doxygen output. This is not specified in the `mif2htm.ini` files, but does appear in files used by Doxygen. Also, the post-processing tools need to know about this.

<!-- begin voy_footer --><!-- end voy_footer --> used for system-wide common footer material. In particular, this can contain the copyright information, date of generation, and document numbers.

Note: The tag names can be changed. However, the change needs to be propagated into `global.pm`, `mif2htm.ini`, the `voyant_master.html` files, and the main header files used by Doxygen (`voyant_head.txt` and `voyant_foot.txt`).

The intent is to be able to quickly update the navigation or copyright information within each HTML file of the system without having to re-generate the HTML files from scratch. This is useful when it is known that the content hasn't changed, but a new date stamp, say, is required in the copyright.

The technique is:

- The HTML files are generated by Mif2Go and Doxygen with these tags where required.
- At generation time, the information between the **begin** and **end** paired tags is empty or minimal.
- The first task performed by the post-processing programs (e.g., voyant_nav.pl) is to locate these tags in the supplied master template files (e.g., `voyant_master_nav.html`). The information between the **begin** and **end** tags from the master file is stored.
- The post-processing program locates these tags in the individual HTML files and replaces information between the **begin** and **end** tags with the information from the master file.
- Post-processing can be performed any number of times (using different master files) without requiring the content to be regenerated with Mif2Go or Doxygen.

An additional tag, **<!-- begin voyant_variables -->**, is created by Mif2Go. It is used to store information about the FrameMaker topic. This HTML comment contains a **##** separated list of paired items (a flag, **##** separator, and content) needed for navigation.

- **##curr##<$$currfile>##** - Flag for the current file and the Mif2Go variable that writes the current filename here.
- **##prev##<$$prevsfile>##** - Flag for the previous file and the Mif2Go variable that writes the previous HTML filename here. Without any post-processing, Mif2Go does not link the first topic of a given chapter to the last topic of the previous chapter.
- **##next##<$$nextfile>##** - Flag for the next file and the Mif2Go variable that writes the next HTML filename here. Without any post-processing, Mif2Go does not link the last topic of a given chapter to the first topic of the next chapter.
- **##firstlast##<$$firstsfile>#<$$lastfile>##** - Chapter first-last flag and the Mif2Go binary variables.
  - **##firstlast##1#0##** - indicates the current file is the first topic of the chapter. This is the flag that adds the topic to the hast table for first topics.
  - **##firstlast##0#1##** - indicates the current file is the last topic of the chapter. This is the flag that adds the topic to the hast table for last topics.
  - **##firstlast##0#0##** - indicates the current file is a topic in the chapter.

Below is an excerpt from the **[Inserts]** section of the `mif2htm.ini` file.

```
; The following three macros in the [Inserts] section
; are intended to each be on one line.
[Inserts]
SplitHead=<!-- begin voyant_variables
    ##curr##<$$currfile>##prev##<$$prevsfile>##next##<$$nextfile>##fir
    stlast##<$$firstsfile>#<$$lastfile>## -->
SplitTop=<!-- begin voy_common_top --><!-- end voy_common_top --><!--
    begin voy_fm_book --><p align="right" class="nav"><a
    href="<$$prevfile>">&lt;Previous Topic&gt;</a> <a
    href="<$$nextfile>">&lt;Next Topic&gt;</a></p><hr><!-- end
    voy_fm_book -->
SplitBottom=<!-- begin voy_footer
    --><hr><center><P><small>PROPRIETARY<br>For Voyant Technology,
    Inc. Internal Use Only.<br>April 2001</small></center></p><!-- gcm
    --><!-- end voy_footer -->
```

Boring Detail: Mif2Go had limitations about providing links to previous or next HTML topics at chapter boundaries (e.g., at the very first topic or the very last topic, respectively.) However, at the end of voyant_nav.pl post-processing, the hash tables do know this

information. As such, those first and last topics are revisited to update their previous and next links to allow sequential browsing through the book over chapter boundaries.

Note: In order for the fixing of previous/next links at the chapter boundaries to work, you have to use chapter INI files and choose file prefixes in a manner that, when UNIX sorted, provides the correct relationship of the FrameMaker chapters to one another.

Chapter Ordering

Mif2Go has a few mechanisms that I use for getting chapters into the correct order, and for assigning topic names that reflect both its content and its order.

- [FileIDs] in the mif2go.ini is used to define a two digit number that separates the contents of each chapter. This prefix is assigned to the all generated HTML and image files. The voyant_nav.pl tool relies on this to sort its input file names and know the order of the chapters. *This is very important*.
- [FileSequence] in the mif2htm.ini is used by Mif2Go in its attempt to provide out-of-the-box topic-browsing over the chapter boundaries. TechPubTools redo that browsing, because Mif2Go still has limitations.
  - » Okay: The Next topic at the end of the chapter is the first topic of the next chapter as defined by the [FileSequence] in the mif2htm.ini.
  - » Not Okay: The Previous topic at the beginning of a chapter is the first topic of the previous chapter as defined by the [FileSequence] in the mif2htm.ini. Readers expect the Previous topic to always be the previous topic. At the chapter boundaries, this means that the Previous topic should be the last topic of the previous chapter and not the first topic of the previous chapter.
- [HTMLStyles] in the mif2htm.ini is where the topic splits are defined and where a portion of the HTML file name is given. In this case, I use the title as part of the name.
- [HTMLStyleFilePrefix] in the mif2htm.ini adds information to the HTML filename that would otherwise be used. In this case, I add the file IDs that were defined in [FileIDs] in the mif2go.ini and a three digit number that indicates the topics position within the chapter.
- [HTMLOptions] in the mif2htm.ini defines the parameters for a built-in variable that counts topics in a file. In particular, I like having the first topic be a 10 and to have increments of 5 in the numbers.

[FileIDs] in the mif2go.ini

Below is an example of the **[FileIDs]** section of the `mif2go.ini` file. Every FrameMaker file name (minus the `.fm` extension) that gets exported from the book is listed. This is followed by the prefix to use on all generated files.

The voyant_nav.pl tool relies on this to sort its input file names and know the order of the chapters. *This is very important*.

```
[FileIDs]
00-tpt-front=00
01-tpt-intro=01
02-tpt-rant=02
05-tpt-environ=05
10-tpt-shell=10
21-tpt-fm-mif2go=21
22-tpt-doxygen=22
23-tpt-dox_filter=23
30-tpt-globals=30
31-tpt-v-nav=31
32-tpt-v-mt-app=32
33-tpt-v-indexer=33
34-tpt-v-latex=34
35-tpt-v-find-extract=35
41-tpt-m-nav=41
42-tpt-m-tree=42
43-tpt-m-index=43
81-tpt-common=81
tp_toolsIX=9x
tpt-title=ac
```

[FileSequence] in the mif2htm.ini

In previous versions of Mif2Go, it could not provide topic-browsing over chapter boundaries at all. The [FileSequence] in the mif2htm.ini is an attempt to provide topic-browsing over the chapter boundaries. However, it is not perfect, because the previous topic at the beginning of a chapter takes the reader to the first topic of the previous chapter instead of the expected last topic of the previous chapter.

Readers expect that if Next takes them always to the next topic in the chapter and then to the first topic in the next chapter and if Previous takes them always to the previous topic in the chapter, then Previous from the first topic in the chapter should take them to the last topic of the previous chapter. In other words, they expect browsing that consistently has a step size of one topic rather than suddenly jumping a whole chapter.

Still TechPubTools implements this so that some form of browsing works right after extraction from FrameMaker and before voyant_nav.pl cleans up the browse sequences.

```
[FileSequence]
00-tpt-front=Front Matter
01-tpt-intro=Introduction
02-tpt-rant=Rant
05-tpt-environ=Environment
10-tpt-shell=Shell scripts
21-tpt-fm-mif2go=Mif2Go
22-tpt-doxygen=Doxygen
23-tpt-dox_filter=Doxygen Filters
30-tpt-globals=Global variables
31-tpt-v-nav=Voyant Navigation
32-tpt-v-mt-app=Voyant Master Tree
33-tpt-v-indexer=Voyant Indexer
34-tpt-v-latex=Voyant Latex
35-tpt-v-find-extract=Find and Extract Code Items
41-tpt-m-nav=Master Navigation files
42-tpt-m-tree=Master Tree Files
43-tpt-m-index=Master Index
81-tpt-common=Common Things
tp_toolsIX=Index
```

[HTMLStyles] in the mif2htm.ini

Readers do not always access an HTML system from the well-defined starting point (e.g., `_start_here.html`). If they happen to be using their Windows Disk Explorer to view the contents of the directory, they should be able to glance at a file name and have some indication of what its contents are, assuming of course that the topic itself has a meaningful title.

[HTMLStyles] in the mif2htm.ini is where the topic splits are defined and where a portion of the HTML file name is given. In this case, I use the title as part of the name.

```
[HtmlStyles]
TitleChapter=Split Title Filename
TitleAppendix=Split Title Filename
TitleFront=Split Title Filename
TitleIndex=Split Title Filename
TitleIntro=Split Title Filename
TitleNoPrefix=Split Title Filename
Indexhead=Split Title Filename
Head1=Split Title Filename
Head2=Split Title Filename
Head3=Split Title Filename
PageLast=Delete
```

[HTMLStyleFilePrefix] in the mif2htm.ini

Whereas I previously stated with [HTMLStyles] in the mif2htm.ini that reader-friendly HTML file names are important, context is also important. By adding a number prefix to the already reader-friendly

file names, the relative position of the topic within the manual and within the chapter can be ascertained at a glance. Moreover, the sorting by name automatically sorts the files into their order in the manual.

I add the file IDs that were defined in [FileIDs] in the mif2go.ini and a three digit number that indicates the topics position within the chapter to the names specified by [HTMLStyles] in the mif2htm.ini.

Three digits as defined by %0.3d may be overkill for most chapters, but it could support a chapter that had close to 197 topics (assuming a starting point of 10 and an increment of 5.)

```
[HTMLStyleFilePrefix]
*=<$$_fileid><$$_splitnum as %0.3d>
```

[HTMLOptions] in the mif2htm.ini

The goal is to have reader-friendly HTML file names which support a name sorting of the files that also reflects the ordering of the topics in the manual. Moreover, whatever naming scheme is used, it should be flexible.

True, I could start my topic numbering at 0 and increment by 1 within the chapter. However, by starting the topic numbering for a chapter at 10 and having the increment be 5, other topics could be inserted manually at later points in time.

Although I realize that as long as I have the FrameMaker source, I'll always insert the topic there and re-export.

The real issue is that one day I might lose the FrameMaker source. The readers probably don't have the source.

If all that I or my readers had was the HTML files, it would not be an insurmountable task to create manually new topics and have them be fully integrated where they belong. Additional tasks would be to:

- update previous / next links on surrounding topics
- update `m_tree.script` files with the new topics
- update index HTML files with any appropriate entries.

```
[HTMLOptions]
...
SplitNumStart=10
SplitNumIncrement=5
StartingSplit=No
```

Index Tokens

My faith in going with Mif2Go as the export tool was well-founded. OmniSys's support and attentiveness to my requests is commendable.

Their support of index tokens is a case in point. Their initial support required generating HTML suitable for Microsoft HTML-Help.

- This inserted MS Object tags with the key word information.
- The Object tags were at the end of the top rather than at the location where the index marker was defined.

I had to create some work-around in my tools to convert the object tag into another format (so that it wouldn't be seen by the reader). My index was prevented from giving my readers mid-topic jumps, which can be helpful in long topics where the writer added index tokens as aids.

The Mif2Go solution is more general that what I required. How I employ their solution is now more easily supported by voyant_nav.pl when it generates the index files. Moreover, it supports mid-topic jumps immediately after export from FrameMaker without any modification to the tags.

Specifically in the `mif2htm.ini` file, I define the following information:

```
[Markers]
Index=VoyIndex

[MarkerTypes]
VoyIndex=Code

[MarkerTypeCodeBefore]
VoyIndex=<a name="<$$objectid>" class="v_index" value="

[MarkerTypeCodeAfter]
VoyIndex="></a>
```

The result is an anchor tag in the HTML file that can be used as a target for a mid-topic hyperlink.

```
<a name="<$$objectid>" class="v_index" value="index entry"></a>
```

The **<$$objectid>** actually comes from FrameMaker that Mif2Go uses and uniquely identifies the paragraph format. The index entry was text that was extracted from writer-defined index token in FrameMaker.

The voyant_nav.pl Perl program can easily locate anchor tags. When they are determined to be of **`class="v_index"`**, it then knows how to handle the name attribute as part of a fully qualified URL to the HTML filename and target within the file (**`00000FileOwner.html#$$objectid`**), as well as the text to display to the reader ("**`index entry`**").

Note: For this to work properly, index tokens should have only one entry per token and should have no more than two levels.

Notes:

Chapter 4 Doxygen

Doxygen is an open-source tool used to extract code prototypes and specially flagged code comments from C / C++ source code as well as Java and IDL (Corba, Microsoft and KDE-DCOP flavors). Doxygen creates an HTML system covering the code items.

In addition, Doxygen is also used imperfectly on Perl files and on an in-house Pascal-like programming language (IVE) files. This required the use of input filters which fake out Doxygen in what it sees and recognizes.

For more information about Doxygen, refer to www.doxygen.org.

Although this chapter is specific to Doxygen, much of what the TechPubTools perform can be applied to JavaDoc output. JavaDoc is another open-source tool that accomplish many of the same things for the Java language; JavaDoc output could easily be incorporated into this solution.

Preparation for Using Doxygen

In order to achieve all of the benefits that Doxygen can offer, you will need support and buy-off from your software engineering organization. They will have to:

- Change their code commenting standards to allow comment extraction.
- Give the technical writer write-access to their source code.

A template will be needed that defines the minimum amount of information and how it should be laid out. Consistency is important in API documentation.

Because the technical writer has much to gain by having Doxygen incorporated into the development process, it behooves them to volunteer for much of the work in first getting Doxygen employed. It is easier to get buy-off of others if they aren't the ones doing the work.

The technical writer needs to:

- Assist defining the template that is to be used in the source code for comments. Remember that the engineers own the code.
  - » Be flexible in the comment styles (//! versus /** ... **/)
  - » Be flexible in the ordering of information in the templates.
  - » Don't give in on things (like @ingroup) that improve the usability of the generated HTML to a wider audience.
- Make a pass through all exposed source files to change existing code comments into the new format.
- Make passes through the code at the engineer's request to edit their comments.

The Doxygen Project File

The Doxygen project file (or configuration file) is an ASCII text file that defines among other things:

- Where to look for source files.
- What directories, files, and file extensions to include.
- What type of output to generate.
- Where to generate output files.

The **tp_tools.dox** file is a Doxygen configuration file that shows how we use many of its features. The features that are of special interest are:

- Input Filters
- HTML Output and Header/Footer Files
- PDF Output

Input Filters

The input filters are a powerful way to change the source files as they are read into Doxygen. The **FILTER_SOURCE_FILES** tag and the **INPUT_FILTER** tag in the project file can be used to specify a program that should be invoked to filter for each input file. Doxygen uses the output that the filter program writes to standard output.

The TechPubTools use input filters to:

- Change comment styles from `//!` into `/**...**/`.
- Change class definitions into a format that is more standard for Doxygen reporting.
- Change the **`@bug`** Doxygen command into **`@lim`** command that is defined in the project file in the **`ALIASES`** section.
- Change program files (e.g., Perl, IVE) into a format that is more or less recognized by Doxygen.

Refer to Chapter 17, "Input Filters to Doxygen," for more information on Perl programs available.

HTML Output

Doxygen does a good job of creating an HTML system for the source files in its project. The **`GENERATE_HTML`** and the **`HTML_OUTPUT`** tags are specified in the project file to tell Doxygen to generate HTML output.

In addition, the **`HTML_HEADER`** and **`HTML_FOOTER`** tags specify two files, in our case **`voyant_head.txt`** and **`voyant_foot.txt`**, respectively. These files contain valid HTML syntax for code that is to be placed at the top and bottom of the generated HTML. In addition, they insert HTML comment tags which are used later by voyant_nav.pl.

`<!-- begin voy_common_top --><!-- end voy_common_top -->` used for system-wide common navigation.

`<!-- begin voy_dox --><!-- end voy_dox -->` used for navigation that is specific to the Doxygen output.

`<!-- begin voy_footer --><!-- end voy_footer -->` used for system-wide common footer material. In particular, this can contain the copyright information, date of generation, and document numbers.

PDF Output

Doxygen does not generate PDF files directly. Instead, it generates LaTex files at the same time that HTML is generated.

The **`GENERATE_LATEX`** and the **`HTML_OUTPUT`** tags are defined in the project file so that Doxygen generates HTML output.

Then the 45_latex_build.b shell script:

- Calls **`voyant_latex.pl`** which creates appropriate **`refman.tex`** and **`doxygen.sty`** files for each **`zlx_`** directory in the project.

- For each **zlx_** directory in the project,
  - Calls LaTex several times in order to generate the table of contents, the index, and resolve cross-references.
  - Creates a postscript file for each.
  - Creates a PDF file.

Doxygen does support a **LATEX_HEADER** tag that can be used to specify a personal header for the generated document. The header should contain everything until the first chapter. In order to accomplish this, you have to be a LaTex expert. If I would have known more LaTex, maybe **voyant_latex.pl** would not have been required.

The PDF output (at the time of writing) does have some limitations.

- The PDF output contains everything in the HTML system, sometimes with a bit too much white space and redundancy (which is useful in the HTML system). In the PDF, it starts looking like filler rather than important content.
- The PDF files from Doxygen / LaTex often have large page counts. This necessitated warnings in the HTML system about *not blindly printing PDF files.*
- In order to change the PDF output and tweak it for your needs, you need to be a LaTex expert.
- When a Doxygen project uses tag files and is dependent on another project, the PDF output is incomplete.

However, the PDF output from Doxygen is still useful. It offers more control in the print layout and in printing page ranges. PDF has its own search capabilities.

Chapter 5 Java TOC Applet

The Table of Contents Applet implements a tree view interface similar to the Win95/NT Explorer. When the applet is enabled in an HTML page for a project (such as m_toc.html), the content of the tree view comes from a script file (such as m_tree.script).

For this TechPubTools project, the input m_tree.script file specified in the Java applet parameters nests individual **`m_tree*.script`** files.

The Java applet that powers the table of contents *cannot* be used and distributed freely in your HTML documentation system.

However, it is affordable. When we ordered it (in January 2002), the source code for the applet and the rights to distribute the applet were $990, I believe. Cheap when I considered how much of my time had already been consumed trying to re-invent this wheel and that my versions were lop-sided and severely lacking in comparison.

Note: *Because the Java applet had been available for quite some time on this website, it took several e-mails to eventually find the owner and arrange payment to make our use and distribution of the applet legal. It was a bonus to get the source code.*

Please refer to www.better-homepage.com/java/java-applets-toc.html or use an Internet search engine with the criteria "java applet table of contents."

Chapter 6 Shell Scripts

UNIX Shell scripts are used to create control the process of generating the HTML system, mostly because this is Voyant's development environment. If we would have had a development environment based on Windows, I would have used DOS batch files.

Simplicity in usage and understanding were the goals in writing these scripts. The naming convention illustrates this most readily.

- All scripts begin with a two-digit number that is intended to reflect the approximate order that the script can be called with respect to the other scripts.
- Information after the two-digit number indicates what operation it performs.
- All scripts have a `.b` extension to indicate (to me) that it is like a DOS batch file that would have a `.bat` extension. Although the extension doesn't play a role in UNIX, it is a useful device to separate file types.
- Assuming that the first digit in the script number is the "series," then a "0" as the second digit in the script number is the master and typically calls the other numbers in the series. For example, 30_tp_tools.b calls the 31_perl.b and 31_script.b files and the 32_perl.b and 32_script.b files.
- Scripts beginning with the same number can be called from the command line in any order, such as 56_nav_script.b and 56_nav_index.b.
- Scripts that have a "5" as the second digit are generally always called from within other scripts and accomplish several operations that would be tedious to repeat in higher level scripts. For example:
  - 35_gen_dox.b is called from 32_perl.b and 32_script.b.
  - 45_latex_build.b is called from 40_latex_build.b.
  - 55_nav_gen.b is called from both 50_nav_update.b and 35_gen_dox.b.

Here is a list of the Shell script files.

- 00_build_tp_tools.b
- 20_cp_com_files.b
- 30_tp_tools.b
  - 31_perl.b and 31_script.b
  - 32_perl.b and 32_script.b
    - » 35_gen_dox.b
- 40_latex_build.b
  - 45_latex_build.b
- 50_nav_update.b
  - 55_nav_gen.b
  - 55_nav_cp.b
  - 56_nav_index.b
  - 56_nav_script.b

Note: Although this TechPubTools project only has one **30_** script file for controlling Doxygen builds of API source code, it is possible to have many of them that output to different **cref_** directories. Moreover, source code dependencies can exist between these Doxygen projects that tag files help resolve. In such an event, the order in which the **30_** script files are called can make a difference and would be specified in the **00_** files.

00_build_tp_tools.b

The `00_build_tp_tools.b` script is intended as the overall control script. If you had to build the entire system from scratch, you could call this one script. Many times, I simply list it (using **more** or **less**) and then pick-and-choose which subsequent scripts to call in the proper order.

When your HTML system is expanded to support additional **30_** script files and **cref_** projects for API source code, this script will need to be updated accordingly

The `00_build_tp_tools.b` script does not call every single other script.

20_cp_com_files.b

This script copies over common files into the top-level `doc_publish` directory. This includes a small number of small GIF files and the CSS file.

30_tp_tools.b

This script is an example of a Shell script that controls running Doxygen. The intent is for there to be a **30_** script and associated Doxygen project for every API source code project.

The modular design helps keep the build process more efficient. The intent is that you only run the **30_** scripts for API source modules that have changed. As such, a valid improvement to these scripts would be the (CVS) commands to check out the source code for this project before running Doxygen.

For simplicity, only one **30_** script file (`30_tp_tools.b`) exists which calls:

- 31_perl.b and 31_script.b
- 32_perl.b and 32_script.b

In larger systems, I use one **30_** script file with associated **31_** and **32_** script files for each major API area.

Note: You are responsible for creating **30_**, **31_**, and **32_** script files appropriate for your API projects.

31_perl.b and 31_script.b

This **31_** script file compartmentalizes the calls required to checkout the source code for this project from our source code repository (CVS). Depending upon the nature of the work, it can vary how often I check-out the source code for a project versus how often I run doxygen against the source (**32_** script files.)

`31_perl.b` checks out the perl scripts.

`31_script.b` checks out the b shell scripts.

If you use some other source control system, you'll have to enter the appropriate commands.

If you don't use source control system at all, you can comment out all references to this **30_** script file.

Note: You are responsible for creating **30_**, **31_**, and **32_** script files appropriate for your API projects.

32_perl.b and 32_script.b

This **32_** script calls the 35_gen_dox.b script for the given project. It also performs some error checking. The purpose of these scripts is to compartmentalize and simplify the running of doxygen against a given project.

32_perl.b runs doxygen against the perl tools.

32_script.b runs doxygen against the b shell script files.

Note: You are responsible for creating **30_**, **31_**, and **32_** script files appropriate for your API projects.

35_gen_dox.b

The **35_gen_dox.b** compartmentalizes and simplifies the several steps that need to be performed before and after running doxygen.

Note: You probably won't need to maintain this file beyond setting up the paths.

This script prepares the destination **cref_** directory by first removing any HTML pages that may have been left by a previous build in both the HTML destination directory (**doc_publish\cref_**...) and the LaTex destination directory (**zlx_**...).

It copies over the common files to its HTML destination directory before starting Doxygen and generating HTML over the top of those files.

It calls doxygen with the appropriate parameters. The Doxygen project file, which by my naming convention has a DOX extension, must be in synchronization with this controlling script with respect to the desired output directories.

Then it calls a tool for generating the **tree.script** file needed for the table of contents.

It runs 55_nav_gen.b to be assured that all HTML topics have the appropriate navigation and that the index file has been generated. This then copies the tree and index files to the **zdoc_merge** location.

When finished, it calls the 56_nav_script.b to regenerate the master table of contents for the entire project. It could conceivably call 56_nav_index.b to regenerate the master index, but this is a slower program with more calculations.

40_latex_build.b

Not all readers appreciate the online HTML system at all times. It requires a computer to read the information, and HTML browsers do not have a lot of control in printing out pages.

PDF files of the documentation system are a useful tool beyond their electronic display and navigation capabilities, because they provide more control in what gets printed out. They can print a range of pages that are laid out more or less as the author intended.

One way that Doxygen achieves its PDF files by first generating LaTex files. From LaTex, the PDF files are created.

This script first calls voyant_latex.pl with appropriate parameters for the project, such as the project file containing version numbers and names. This generates a couple of Latex files that are used later in the build process.

Then this script creates the PDF files from the Doxygen LaTex output by calling the 45_latex_build.b.

Note: When you add **cref_** projects for API source code, this is one of the files that you will manually have to update accordingly. You are responsible for creating **40_** script files appropriate for your API projects. Generally, one **40_** script can cover all projects.

45_latex_build.b

The `45_latex_build.b` compartmentalizes and simplifies the several commands required to create a PDF file from the doxygen Latex output.

The commands provided are from Doxygen.

The Doxygen generated PDF files are not without their limitations.

- They tend to have really large page counts.
- If the source code for the Doxygen project used tag files and had dependencies on other systems, the hyperlinks and related information come up unresolved, incomplete, or missing.

Whereas the second limitation is rather severe, it only affects the PDF files and not the HTML output. It is presently an inconvenience that open-source Doxygen may eventually resolve.

50_nav_update.b

This script calls 55_nav_gen.b or 55_nav_cp.b for all projects in the HTML system.

All directories that are to be included in the system should have a line in this script so that they can be appropriately processed.

There is no requirement that 55_nav_gen.b be called with the same master file for each directory.

A valid argument can be made that the navigation and / or copyright could be different from directory to directory. However, when the same master file is used for all directories, changes can be made in one place and propagated throughout the system.

A more useful variation of this in quickly re-purposing the system is to have multiple **50_** scripts. The **50_** scripts are identical in terms of directories but reference different master files for different purposes. For example, **50_nav_update_internal.b** could specify `master_internal.html` in the 55_nav_gen.b calls for all directories,

while **50_nav_update_external.b** specifies `master_external.html`. In this manner, copyright and navigation could be changed in a matter of seconds between internal and external usage.

Note: When only a portion of your system has changed, it came sometimes be more efficient to view this file first to learn what commands it issues. And then instead of calling **50_nav_update.b** from the command line, you would issue the appropriate 55_nav_gen.b command (followed by the **56_** scripts). This technique is only appropriate for larger systems that were previously generated and when changes are localized.

Note: When you add **cref_** projects for API source code or **book_** directories for FrameMaker manuals, this is one of the files that you will manually have to update accordingly.

My environment has a `techpubs/perl` directory that is parallel to the `techpubs/tp_tools` directory. References from the shell scripts to a Perl program use a relative path `../perl/<program>`. For the purposes of simplicity in creating the ZIP file and clarity in bringing everything together, all scripts have been updated to reflect this new path `techpubs/tp_tools/src_perl/`, which is simply `src_perl/` from the scripts.

55_nav_gen.b

This script calls voyant_nav.pl to swap out navigation bars and copyright information on all HTML files in the selected directory using the specified master file. Refer to the documentation on voyant_nav.pl for more details about what gets accomplished with this script.

When finished, it copies the generated **tree.script** and index files as unique files to the **zdoc_merge** directory for later processing.

There is no requirement that voyant_nav.pl be called with the same master file for each directory.

A valid argument can be made that the navigation and/or copyright could be different from directory to directory. However, when the same master file is used for all directories, changes can be made in one place and propagated throughout the system.

55_nav_cp.b

This is a direct copy of 55_nav_gen.b but with the voyant_nav.pl calls commented out. This way the owning 50_nav_update.b can use the appropriate 55_nav_gen.b or 55_nav_cp.b calls depending upon the needs.

It copies the generated **tree.script** and index files as unique files to the **zdoc_merge** directory for later processing.

Note: Some Voyant projects are unique in how they are generated and do not require the additional voyant_nav.pl calls.

56_nav_index.b

This script performs several operations needed to prepare and finalize the index for the HTML system.

- It removes any leftover index files or master index files from the temporary directory (zdoc_merge).
- It removes any master index files from the final directory (doc_publish).
- It copies default master index files to the temporary directory (zdoc_merge). The default files state that the letter has no entries and when appropriate are overwritten by voyant_indexer.pl.
- It copies from the directories of the system all index files that were generated by voyant_nav.pl and gives them unique names in the temporary directory (zdoc_merge).
- Runs voyant_indexer.pl to create new master index files in the temporary directory.
- Copies all master index files from the temporary directory (zdoc_merge) to the final directory (doc_publish).

A single template file is specified on the command line to voyant_indexer.pl and is used to generate the master index files. However, voyant_indexer.pl only generates index files for letters that it encounters. This can leave holes in the index navigation. Hence, the suite of index files is first populated with defaults for all letters using the **cp** command and the template file.

Refer to the documentation for voyant_indexer.pl to find out all that this script accomplishes in the index that is generated.

56_nav_script.b

This script performs several operations needed to prepare and finalize the master tree (table of contents) for the HTML system.

- It removes any leftover tree files or master tree files from the temporary directory (`zdoc_merge`).
- It removes any master tree files from the final directory (`doc_publish`).
- It copies from the directories of the system all tree files that were generated by voyant_nav.pl and gives them unique names in the temporary directory (`zdoc_merge`).
- Runs voyant_mt_app.pl to create new master tree files in the temporary directory.
- Copies all master tree files from the temporary directory (`zdoc_merge`) to the final directory (`doc_publish`).

Refer to the documentation for voyant_mt_app.pl to find out all that this script accomplishes in the table of contents that is generated.

Notes:

Chapter 7 globe.pm

The **globe.pm** file is a Perl package that contains global variables are routines that are used in several other Perl programs.

Overview

Initially, there was only one program (voyant_nav.pl).

- All variables were defined at the beginning of this program. This facilitated changing the value of external facing tags, such as the syntax of special HTML comment flags that the program needed to recognize.
- Most variables were global to that program, because it reduced memory usage and confusion in naming / meaning when passed into routines.

During the development of the second program (voyant_indexer.pl), many of the variables were for external facing tags were re-used. It became a maintenance hassle to update these variables in multiple places.

A design decision was made to place these global variables into a single location, the **globe.pm** file. The reasoning was:

- Passing these variables by value makes copies of the information in memory.
- Passing these variables by reference is essentially the same thing as using them as a global variable but with potentially additional complications, such as variables changing names.
- The call to the routine becomes more complicated when variables are passed in particularly when considering the significant number of HTML tags that the various routines need to recognize and handle.

- Many of the variables are used in other programs and define tags that are specific to your environment. Placing the definitions in one place makes them easier to change.

Now, many of the Perl programs reference this package file which instantiates the variables. Then they can be accessed by any subroutine in the program. Whenever these are used in another All such global variables are prefixed with "**globe::**".

In addition to the global variables, several routines that are used by more than one program have been relocated to this **globe.pm** file.

Variable and Data Structures

Many of the global variables migrated into more complex data structures consisting of a hashes of arrays. The reasoning was:

- Special code needed to be written to handle each unique variable. The code was rather lengthy, hard to maintain, and hard to copy copied into other programs which might require similar functionality and variable definitions.
- Many of the external facing identifiers which delineated information in, say, a generated HTML file had a start tag and an end tag. This requires two variables for every identifier that I wanted to catch. An array reduces the number of unique variable names by a factor of two.
- Hashes of arrays can reduce the number of unique variable names even further.

By putting more complexity into the data structures, the code became simpler for what was required to fill the data structure, extract elements, or otherwise manipulate the data. Moreover, the code could be re-used and handle situations where tags might be (purposely) missing.

The complex data structures impose standards and conventions. In some cases, once the hash is defined in the **globe.pm** file, the code in the perl (`.pl`) file does not need to know the explicit keys into the hash, which further generalizes routines to manipulate the hashes.

If you do not like some of my tag naming conventions for things that appear in HTML files (but not necessarily to readers), the **globe.pm** file is generally where they can be tweaked for your purposes.

WARNING! Even if the globe.pm file contained only variable definitions, it is a piece of code where syntax does matter. Be careful in making changes to definitions, otherwise other Perl programs won't work.

The variables and data structures in the **globe.pm** file have comments regarding which perl program uses them.

Note: If a given perl program doesn't work, it could be that the hard-coded path to the **globe.pm** *file is invalid for your environment.*

Common Routines

The problem of code re-use and re-purposing became an issue even while developing the first couple of Perl programs, and only became worse the more programs were copied and modified for other applications.

On the one hand, I had to determine what was unique to one application and what was truly general and valid as a shared routine for several applications.

On the other hand, I had new data structures which simplifies code further. Code needed to handle a data structure could be re-used to handle other similarly structured data structures.

As a result, many of the routines were generalized to the point where they could re-used. If two or more programs needed a routine, it was placed into the **globe.pm** file. Not all of the routines are used in all Perl programs.

WARNING! By adding common routines to the globe.pm file, it solidifies it as a code file. Be careful in making changes to definitions, otherwise other Perl programs won't work.

Notes:

Chapter 8 voyant_nav.pl

The **voyant_nav.pl** perl program is the central tool in the TechPubTools suite around which all of the other shell scripts and perl programs rely.

Overview

The **voyant_nav.pl** perl program does most of the work in creating a comprehensive HTML system that spans the mini-HTML systems generated by Doxygen and Mif2Go (from FrameMaker source).

Specifically, this tool:

- reads a master file which specifies information for the head, top of the topic, and bottom of each topic.
- swaps out information from a master for each topic and removes certain HTML tags specified by the master (for better CSS control).
- creates several hash tables that ultimately determine topic order, topic level in tree (table of contents, TOC), etc.
- generates a mini-Table of Contents file (tree.html and tree.script) for the directory. This file is then re-used and combined with others later to create the comprehensive system.
- processes index tokens coming from the output of Mif2Go.
- creates index tokens from the topic hash table.
- parses index tokens generated from Mif2Go or already existing in the file.
- writes index entries to `_index_file`. This file is then re-used and combined with others later to create the comprehensive system.
- figures out the previous-next topic browsing.

The Beginnings

The **voyant_nav.pl** perl program started out as a tool just to swap out the header, navigation, and copyright areas. These were items that needed to remain consistent for all generated HTML files in a subproject.

The goal was to have a tool that could post-process HTML files and, say, update the copyright area at the bottom of each file with newer information without having to go back to the (unchanged) source and re-export.

Likewise, the header and navigation areas were items that could frequently change based on trial-and-error of the look-and-feel and the whims of my audience. Again, there was no sense re-running extraction programs against the source if the general content remained unchanged.

To keep the **voyant_nav.pl** program manageable, it focuses only on the HTML files in one directory. A shell script (such as 50_nav_update.b and specifically 55_nav_gen.b) can conveniently call this program with the specific input files for the project.

The **voyant_nav.pl** program reads in information from external master files in order to avoid hard-coding data that frequently changes or could potentially change, such as the tags to look for in the HTML files.

The Extensions

The index and table of contents are two special areas which theoretically could be created with their own tools. Such tools would require opening and scanning each file in the input directory in order to locate pieces of information of interest.

Because the **voyant_nav.pl** program already opens each file, reads it into memory, and scans it for information of interest, it was enhanced to tackle the mini-index and mini-table of contents just for the files in its directory.

CYA

The **voyant_nav.pl** program was built up over time. Programming was done iteratively and piecemeal. Debugging statements were liberally created and then were initially deleted once the section of code seemed to work.

As the complexity of the program increased, many of the same debugging code had to be re-entered to help trace the operation and verify the results at various stages. Over time, I stopped deleting debug sections and instead conditioned them out in order to keep them available when needed later.

An early CYA effort was to always write a `_temp` file of the original HTML file. This way the input could be compared to the output. Then in one line (that could be turned on or off) the program overwrites the original input HTML file with the contents of the `_temp` file.

ultimately overwrites the HTML file that it reads it. However,

Data Structures

The **`voyant_nav.pl`** program initially had simple variables.

When other tools were needed, such as voyant_mt_app.pl to create the master table of contents or voyant_indexer.pl to create the master index, I discovered that many of the same variables and routines were required.

I placed global variables and frequently used routines in the **`globe.pm`** file, a Perl package.

More complex data structures were created later as the program grew, because they simplify the program execution, make it more reliable, and allow for code re-use.

Topic Browsing

Another of the extensions of the **`voyant_nav.pl`** program was to create previous-next topic browsing that crossed over chapter boundaries or to simply implement previous-next topic browsing.

- A limitation of our off-the-shelf extraction tool from FrameMaker was that it could only offer browsing within a chapter.
- A limitation of our off-the-shelf source code extraction tool was that it had no browsing.

In order to allow browsing over chapter boundaries from the FrameMaker, I made restrictions on the naming convention of the generated HTML files. The naming convention is:

- a two-digit number for the chapter [required], followed by
- a three-digit number for the topic in the chapter, followed by
- plain text regarding the topic title [required to be reader-friendly].

In this manner, all files for a given chapter are grouped together by the prefix; all topics are placed in order when sorted alpha-numerically; and all files have meaningful names.

The grouping of files by chapters (first two-digits) is the most important aspect of the naming convention, because when it allows the brain-dead **voyant_nav.pl** program to know the order of the chapters from the name.

Specifically, each HTML file extracted from a FrameMaker file knows whether or not it is the first topic in the chapter, the last topic in the chapter, or a topic in the middle of the chapter. Information about the first and last topics is stored in a hash. Once all of the files in the directory have been processed, all first and last topic files are re-visited so that their previous and next links can be updated with information that is known about the last topic of the previous chapter or the first topic of the next chapter.

In the case of code files (which have no topic order tags), their names determine the order. Once all of the files in a code directory have been processed normally, they are revisited so that their previous and next links can be updated with information.

Index Tokens

When pulling together multiple manuals or mini-HTML documentation systems, the most effective and easily understood way of implementing cross-references between manuals or systems is to have an effective index.

Two extensions of the **voyant_nav.pl** program were to have it generate index entries for the output file based on content (e.g., <H1> tags) if none existed, and to have it handle index tokens that might be inserted by other programs, such as Doxygen or Mif2Go.

Doxygen Index Tokens

In the case of the Doxygen extraction from source code, the generated HTML files had many anchor tags of the format:

```
<a name="a0" doxytag="dox_comment_chg.pl::comment_count"></a>
```

These anchors were used as mid-topic jumps sometimes from within the same (often very long) HTML file and from other HTML files. The **doxytag** extension was an easy tag to spot. Moreover, value of the **doxytag** tag was useful information for the index.

Hence the **voyant_nav.pl** program was enhanced to locate these tags and extract relevant information for the entries into the index file.

Mif2Go Index Tokens

As was previously mentioned, my faith in going with Mif2Go as the export tool was well-founded particularly when it came to OmniSys's responsiveness to my request for better index support.

My proposal to OmniSys used the Doxygen **doxytag** anchors as a model. Once implemented and supported in my `mif2htm.ini` files, I could create anchor tags of the format:

```
<a name="<$$objectid>" class="v_index" value="index entry"></a>
```

The **<$$objectid>** actually comes from FrameMaker that Mif2Go uses and uniquely identifies the paragraph format. The index entry was text that was extracted from writer-defined index token in FrameMaker.

The voyant_nav.pl perl program can easily locate anchor tags. When they are determined to be of **class="v_index"**, it then knows how to handle the name attribute as part of a fully qualified URL to the HTML filename and target within the file (**00000FileOwner.html#$$objectid**), as well as the text to display to the reader ("**index entry**").

Hence the **voyant_nav.pl** program was enhanced to locate these new **v_index** tags and extract relevant information for the entries into the index file.

Note: For this to work properly, index tokens should have only one entry per token and should have no more than two levels.

Input

voyant_nav.pl takes the following parameters:

path - location to find the HTML files. The name should be terminated with a slash (\). Although the directory_name is the first command line parameter, the true input are the HTML files contained within that directory.

master_nav_file - [optional path and] filename for the HTML file to use as the master for information. This file has several specially flagged HTML comment sections that are required. Information from the tagged sections are lifted and placed into the tagged sections of the input HTML files.

Output

voyant_nav.pl returns or modifies the following:

input_html_file is modified with new information in the tagged areas. This includes the information in the <head>, navigation at the top of the <body>, and copyright information at the bottom of the <body>.

_file_list is a temporary file with a list of all HTML files in the given directory.

_index_list is a file with extracted index tokens.

tree.script is a file with a mini-TOC (table of contents) for the given book.

tree.html is a temporary file with a mini-TOC (table of contents) for the given book.

Notes:

Chapter 9 voyant_mt_app.pl

The **voyant_mt_app.pl** perl program refers to the "main tree (mt)" that plugs into the java table of contents (TOC) applet. This assumes that `tree.script` files were created for all sub-projects by voyant_nav.pl, which itself is usually called from the 56_nav_script.b shell script.

Using all available and uniquely named `tree.script` files, this creates a series of master `m_tree*.script` files which are nested inside a master `m_tree.script` file.

Overview

This program reads in the project file. Based on the directory names, it makes some assumptions about the names of the tree script files that it expects to see. It steps through all tree script files, reads them in, creates associated master tree script files, and then creates a master tree script file that nests the other tree script files in order.

If an expected tree script file is missing, an error message is output but a working master tree is still generated that ignores the missing mini-TOC.

Note: *When the original tree script files were created by voyant_nav.pl, they had no path information in the hyperlinks. This was to make them more general and more easy to use in other directory structures. The tree script files needed to stand on their own and be independent of any resulting directory structure where they might be shared.*

Hence, this program must create master script files that include the expected relative paths to the destination topics. The relative path information is again the directory names from the project file.

The Beginnings

The **voyant_mt_app.pl** perl program started from another home-grown program called **voyant_mt_tree.pl**. That program used HTML files (tree.html) created for the mini-TOCs from voyant_nav.pl and then generated a series of HTML file. It created expanding/collapsing topics for only the top two levels (chapter and heading1).

Although I liked the HTML files because they are fast to load and simple, it was limited to only expanding the top two levels. As a technical writer, I did not have the time to spend programming a more general solution. Moreover, the multiple HTML which provided "fake" expanding/collapsing topics would soon become unmanageable.

The Extensions

After searching the Internet, an even better solution was found that is more flexible, better, and cheaper than anything I could come up with. It is a Java applet that implements the table of contents.

The Table of Contents Applet implements a tree view interface (similar to the Win95/NT Explorer). The two features that impressed me the most (aside from the inexpensive price for the source code) were the simplicity of the input scripts and the ability to nest scripts.

Refer to www.better-homepage.com/java/java-applets-toc.html.

Once I had convinced management to let me spend the money, it was easy to modify these tools to support the script files.

Input

The **voyant_mt_app.pl** perl program takes the following parameters:

path location to find the tree files. The name should be terminated with a slash (\). Although the directory_name is the first command line parameter, the true input are the tree files contained within that directory. Tree files must begin with "tree_".

project_file [optional path and] filename that defines all directories, their titles, and their sequence in the comprehensive HTML system.

Master tree files are another way of saying Table of Contents files. Tree files themselves are mini-TOC files that were generated by voyant_nav.pl.

Output

The **voyant_mt_app.pl** perl program creates **m_tree_** script files for each directory of the system. The files are named such that they are associated with their owning directory. In addition, it creates a single master **m_tree_** script file that nests the other script files.

Notes:

Chapter 10 voyant_indexer.pl

The **voyant_indexer.pl** perl program uses the temporary files for the index and generates a series of HTML files that are the master index over the whole system.

It can also perform word-chunking that expands the number of index entries and turns it into more of a concordance, a useful feature for programs.

This assumes that _index_file files were created for all sub-projects by voyant_nav.pl, which itself is usually called from the 56_nav_index.b shell script. Using all available and uniquely named _index_file files, this creates a series of master m_idx*.script files.

Overview

This program issues a system call to create a list of candidate input **index_** files in the **path** directory. Then it steps through each of those files and concatenates their input into **master_raw**.

master_raw is turned into a hash table **master_index**. The key into the hash table is the index entry. What gets stored is both the title and the URL.

Word-chunking, when turned on, is performed on each element in the **master_index**. Natural boundaries (spaces, dashes, underscores, changes in case in the middle of a word) are used to create additional two-level index entries that contain the word-chunk followed by where it came from.

The **ignore_terms_file** is used to eliminate useless word-chunked entries (such as “the”, “a”, “to”, etc.) The additional useful entries are appended to the contents of the hash **master_raw** using the **division_mult_entry** separator only if the new entry is not a duplicate.

Word-chunking is particular useful for API documentation so that the reader does not have to remember the exact name of a code item in order to find it. An initial index token of "`api_GetMovie-list`" could be found not just under its name in the "A's", but under "get", "movie", and "list".

The expanded list is sorted.

The sorted list is output to a series of **`m_idx_`** files. New **`m_idx_`** files are created whenever a new character starts a word in the list.

When an index entry is referenced by multiple URLs, the additional references appear in the output as small document icons next to the first reference in plain text.

The Beginnings

Originally, I had mini-HTML systems that came from FrameMaker through Mif2Go and from source code through Doxygen. I was in charge of the system, so could manipulate the source documents to create links between mini-HTML systems.

The problems with this technique were:

- Such manual hyperlinks are inflexible to changes in the directory structure.
- Limitations in certain tools required more hard-coded paths than I wanted to have in my hyperlinks.
- Doxygen in particular does not like hyperlinks which are fully qualified and tended to break them.
- The mini-HTML systems could not be used out-of-context in other situations where they might be shared (e.g., another documentation suite), because there were too many interdependencies between systems.
- The inter-system dependencies were hard to maintain and keep working, particularly when the overall structure of the system was not yet set in stone.

The Extensions

How do printed manuals allow you to cross-reference between them? What if the manual name changes? What if the cross-reference target changes?

It seems to me that technical writers often purposely do not add cross-references to other manuals. One reason is the problem of maintaining the references. However, another reason is that you don't have to cross-reference everything.

I almost developed an extension tool that found created database references of topics (such as titles, code item names, etc.) and the HTML file where they were located. This tool would then go through the entire system and automatically add hyperlinks.

I stop myself from wasting time developing this because:

- It is difficult to tell a tool when enough is enough. Every page would be full of hyperlinks.
    - This would be hard to read and follow.
    - This would distract the reader and lead them astray when really the writer intended for them to stay on that subject before moving on.
- It is easy to get one-to-one hyperlinks, but more of a challenge to get one-to-many hyperlinks, which is what the case would be if you wanted to web all occurrences of some topic or keyword together.
- It occurred to me that when something is of interest in a printed manual, the technical writer makes an index entry.
    - For multi-book documentation suites, the technical writer may have a master index covering all manuals.
    - Even if there is no master index, each manual has an index. As a result, the technical writer can rely on the reader's intelligence to go looking in the indices of the various manuals (if the technical writer didn't already point them to the right manual) to find more information.

As such, I curbed my desire to program more by fleshing out the master index, which in my case is modular and always covers the suite of documentation in the project.

Because the index files are created individually for the sub-projects, those sub-projects can be shared in other documentation suites and not mess up the master index.

True, it would be nice on occasion if I could put in links between mini-HTML systems. However, if the mini-systems are indexed properly, the reader can get where they need to be.

Building on the Java TOC Applet

Although the Java TOC Applet is used for the table of contents, it could theoretically be used for the index, as well.

I explored this direction for a short time with the voyant_indexer.pl perl program. It still outputs **m_idx.script** which can be plugged into the Java TOC Applet (such as on a **m_idx.html** file) in the same manner that the m_tree.script files are plugged into the m_toc.html file.

I stopped pursuing this because:

1. Some of my projects create large indices on the order of 20,000 entries or more, depending upon whether or not word-chunking is turned on. It may have necessitated breaking the index into nested script files by letter, much like the table of contents, which introduces further loading delays.
2. The script files cannot be searched from the browser. Therefore, it can be more difficult to locate entries in a large index even after expanding it.
3. The window for the applet (even if put into a separate browser instance) place limitations on the display region and scrolling.
4. Most important of all, the present design of multiple HTML files for the index works; the pages load fast, scroll easy, and can be searched.

Input

The **voyant_indexer.pl** perl program takes the following parameters:

path location to find the index files. The name should be terminated with a slash (\). Although the directory_name is the first command line parameter, the true input are the index files contained within that directory. Index files must begin with "index_".

master_tree_file [optional path and] filename for the HTML file to use as a template for all index files to be generated. Ideally, this should contain navigation tools to get between the generated index files [a-z] and other parts of the system, such as the table of contents. This file has several specially flagged HTML comment sections that are required.

ignore_terms_file [optional path and] filename for a text file that contains words to ignore in the word-chunking process.

The true input are the "**index_**" files. These files are an unsorted running list of index tokens that were extracted from the HTML files in a directory. The tokens have two parts: the index entry and its URL. The separator is defined in the `globe.pm` file and is **:,:** (**$globe::word_url_boundary**). Additionally, the index entry can have two levels. In such cases, the separator is **:;:** (**$globe::word_c_boundary**). Finally, a given index entry can represent multiple references or URLs. In such cases, the multiple entries are separated by **:;;;:** (**$globe::division_mult_entry**)

If the separators are changed in the generator program (voyant_nav.pl), they need to be changed here, too. The variable names in both programs are the same. The separators themselves were chosen because they were deemed never to occur in an index entry or URL and aren't Perl special characters.

More information about the **master_file**. Aside from serving as a template for all generated index files, this file chunks information using specially tagged HTML comments in order to simplify locating where generated information is to be placed. In addition, some tags contain information critical to the proper operation of the indexer.

This does not support index entries that might be or have Perl special characters. These are often eliminated early in the process.

The input **index_** files cannot have "`.htm`" as part of the name. This assumes that input information was of the proper format with an index entry, **$globe::word_url_boundary**, and URL. If any of the input **index_** files did not have this, this can cause problems.

Output

The **voyant_indexer.pl** perl program creates a series of HTML files that begin with "**m_idx_**". Generally, what follows in the name is the first character of the first word within the file. All index entries beginning with that character are in that file. These files are created in **path**.

In addition, **voyant_indexer.pl** perl program outputs **m_idx.script** which can be plugged into the Java TOC Applet.

Notes:

Chapter 11 voyant_latex.pl

voyant_latex.pl changes two Latex files in each zlx_ directory of the system so that they contain the appropriate information for the project before the PDF files are generated. Most of the content for the PDF files comes from Doxygen and is in a Latex format.

Input

path default location to find the Latex temporary directories. The name should be terminated with a slash (\).

This should be the top directory where various **master-doxygen.sty** and **master-header.tex** can be found and more importantly, the **zlx_** directories where generated LaTex files from Doxygen reside.

master_project_file [optional path and] filename for the file that stores the information needed for the project and the latex files. Ideally, this should be the same project file used by the voyant_nav.pl tool, because this contains the **voy_order** for the project.

latex_variables [optional path and] filename for the file that contains additional comment tags with the **voy_latex** variables and the **voy_latex_head** variables.

Global variables are defined in the **globe.pm** file, a Perl package. Whenever these global variables are referenced here, they are prefixed with "**globe::**".

Each Doxygen directory in the project file can have their own directories with LaTex files. The master Latex files in each of those directories can be updated with appropriate information so that the headers and footers in the PDF file contain relevant information.

To do so, several Latex variables are defined in the **<!-- begin voy_latex -- !-- end voy_latex -->** comment of the **latex_variables** file. They are:

```
"##master_header_tex##", "master-header.tex"
"##master_doxygen_sty##", "master-doxygen.sty"
"##output_header_tex##", "voy-header.tex"
"##output_doxygen_sty##", "doxygen.sty"
"zz-yyyy-by-companyname-zz", "by Voyant Technologies, Inc."
"zz-genbydoxversion-zz", "Generated by Doxygen 1.2.11.1"
"zz-outside-lfooter-zz", "Voyant Technologies, Inc."
"zz-inside-rfooter-zz", "zz-docnum-zz"
"zz-inside-lfooter-zz", "zz-docnum-zz"
"zz-outside-rfooter-zz", "zz-doctitle-zz"
```

Another Latex variable can be defined in the **<!-- begin voy_latex_head -- !-- end voy_latex_head -->** comment of the **latex_variables** file. It is:

```
"refman.tex", "\begin{titlepage}", "\end{titlepage}"
```

Output

voyant_latex.pl generates appropriate **header.tex** and **doxygen.sty** files for each **zlx_** directory in the system. These are then used in Latex's PDF generation for each subproject. It first reads the reads in the **voyant_master_nav.html** file (or equivalent). This file must specify:

- The **voy_order** for the system that lists all directories, associated names, and associated PDF files.
- The **voy_latex** which contains variables used in LaTex build such as master file names, company name, etc.
- The **voy_latex_head** which contains additional information for the **refman.tex** file.

voyant_latex.pl steps through all **cref_** directory names in the **voy_order** and generates an equivalent **zlx_** directory name. These directories should be found under **$globe::path**.

The **voy_latex** variables declare where additional **master.tex** and **master.sty** files reside. These contain call-outs for variables that **voy_latex** resolves for the given project. All variables in the master files are resolved with information appropriate for the project.

Upon conclusion, appropriate **refman.tex** and **doxygen.sty** files are written to each **zlx_** directory.

Chapter 12 find_extract.pl

The **find_extract.pl** perl program is a handy tool for API applications that would tend to contain too much noise. It generates a temporary file from many source code files. The temporary file contains only those code items of interest.

Overview

In more general terms, the problem was that the engineers' working directories often contained more code files and code in general than what was required by down-stream API users. When I ran Doxygen on these files and directories, it would pick up many more code items than were useful and significantly added noise to the output.

- For one API project, I had C++ classes which implemented "functions" to an internal Pascal-like programming language.
- For another project, all sub-projects created and exposed to all users a small set of functions. These functions were distributed throughout the code and were surrounded by the C++ code that made them possible. (These were called **xhelp** functions, which is why the **x** creeps through in implementation and descriptions.)

From the onset, this tool needed to support different requirements in terms of what was passed through to the temporary file.

Hence, each project can define its own **xscope**, which is a perl module that limits it definitions to the contents of several data types that are used later.

Input

input_scope A file path and name to a perl package that has the to-be-included and to-be-excluded code items commands.

root_path_to_files must be terminated with a forward (\ /) slash. If output file does not have a forward slash (\ /) -- an indication of a path --, then the <root path to files> is assumed.

output_file the name of an another file. This file, plus one similarly named with a leading underscore are generated. The file without the underscore is intended as the input to Doxygen.

The **input_scope** file is the key to the successful operation. It is required to have:

- The **xscope** perl package definition.
- Routines
  - **declare_variable**.
  - **memory_clean_up**.
- Arrays:
  - **x_names**, which contains all prefixes that could be of interest.
  - **needed_in**, which contains a list of xhelp commands that are absolutely needed-to-be-in.
  - **needed_out**, which contains a list of commands that we don't want to expose at all if they happen to come through.
  - **include_f_type**, which are file types that should be viewed.

Output

In short, the **find_extract.pl** perl outputs these files:

<output_file>.gen which has part of its name specified as an input parameter. It contains the located code item and all associated doxygen comments. This has only a prototype definition; the actual code body for the code item is empty.

Example: When global (**xhelp**) functions were dispersed around the code pool, this file would put the function prototype and its associated comments in this file.

<output_file>_class.gen which has part of its name specified as an input parameter. It contains the definitions of any classes, their member functions of interest, and their associated comments. This has only a prototype definition; the actual code body for the code item is empty.

Example: Sometimes developers have classes with many member functions, but only several of which that they want to expose in the documentation. This file would contain only those classes and member functions deemed worthy of exposing. It saves the developer from having to play tricks with private and public declarations. Moreover, in my case, the exact member function name was not actually exposed, but something similar.

<output_file>_dox_template.gen which has part of its name specified as an input parameter. It contains just the associated doxygen comments.

Example: For one project, developers implemented a Pascal-like language using C++ classes. The Pascal code that was exposed to users was generated. The appropriate place for the documentation of the end-user functions was on a class member function definition. These comments needed to be found and extracted. I used the same code generation tools as the developers to generate a different temporary file with the function definitions that matched what came out of the extracted doxygen comments. Both files were then used as input into doxygen.

<output_file>_list.gen which has part of its name specified as an input parameter. It contains a list of located code items and a checklist whether or not they had doxygen comments.

Example: This file was used mostly as communication between Technical Publications and Engineering. It showed which code items I found and whether or not they were commented as expected. Engineering was expected to review the list, remove code items that weren't supposed to be picked up, add code items that were, and comment those items that were flagged as not having templates.

Depending upon the application, any of the files except the **<output_file>_list.gen** file could be used as subsequent input into doxygen.

Note: The output files used as input to Doxygen have only prototype definition; the actual code body for the code item is empty. Doxygen doesn't care about the implementation details below the definition of the code item, and neither does the API reference documentation.

Implementation Details

The **find_extract.pl** perl program does divide and conquer.

1. The **find_extract.pl** perl program greps the code
   - Using the prefix list and looking at the file types of interest.
   - Using the command list and looking at the file types of interest.
   - This information is placed into a very temporary file.
2. The temporary file with grep results is stripped of non-interesting entries.
3. A file hash is created that is a hash of hashes.

```
$file_hash {$src_file} {$code_item} {r} {$return_type} = unimportant number
$file_hash {$src_file} {$code_item} {p} {$prototype} = doxygen
```

It contains:

{$**src_file**} the source code files where items of interest were found.

{$**code_item**} code items of interest, each associated with a source file.

{r} {$**return_type**} a hash of return types for each code item; supports overloading.

{p} {$**prototype**} a hash of code definitions for each code item; supports overloading. It contains to the doxygen comment block.

4. The source files from the hash are
   - opened and searched for their respective code item definitions.
   - are searched for comment blocks associated with the code item definitions.
5. The file hash is
   - fleshed out for the prototype definitions and their respective comment blocks.
   - is stepped through and output to generated files. The generated files are intended for input to doxygen.

The final step is the temporary output files are generated easily based on the information in the hash tables. The output files have only prototype definition; the actual code body for the code item is empty. These files are used as input to Doxygen. Doxygen doesn't care about the implementation details below the definition of the code item, and neither does the API reference documentation.

Chapter 13 tree_js_2_script.pl

The **tree_js_2_script.pl** perl program is used in conjunction with Doxygen to create a `tree.script` file from a Javascript file.

Overview

Doxygen has a configuration option called **GENERATE_TREEVIEW**. When this is enabled, it creates two files:

- **treeview.js** is a general Javascript application that is referenced in the Doxygen output. It contains the routines required to have folders and topics that provide an expandable / collapsable table of contents in a left-hand navigation frame.
- **tree.js** is a Javascript file with data specific to the project. The `treeview.js` acquires its expandable / collapsable content from the `tree.js` file.

Under normal circumstances, the `treeview.js` and its associated `tree.js` data could have been re-used and re-purposed in the other settings, instead of searching for another solution, such as the Java TOC Applet.

I did have some experimental tools that could combine the `tree.js` files from several sub-projects into a comprehensive one. The problems were that the Netscape Navigator didn't like the `treeview.js` at all and that Microsoft Internet Explorer would issue memory warnings as the `tree.js` content increased.

Still, the overall structure of the `tree.js` file from Doxygen was commendable. When the Java TOC Applet was discovered and its script files researched, a design decision was made to simply convert from the Javascript format needed for `treeview.js` into a script format needed by the Java TOC Applet.

The **tree_js_2_script.pl** is usually called from within the 35_gen_dox.b shell script, right after it performs the Doxygen build.

CAUTION! ***The tree_js_2_script.pl perl program is heavily dependent on Doxygen and the assumptions and formats in the output tree.js file.***

Input

The **tree_js_2_script.pl** perl program takes the following input arguments.

path_to_tree the directory wherein the tree.js files can be located.

name_of_input a relative path and name for the input javascript file. This is usually the tree.js file.

name_of_output [optional] a relative path and name for the output script file suitable for the Java TOC Applet. If this is not provided, the **name_of_input** is used but has its extension changed to .script.

The **tree_js_2_script.pl** perl program requires access to **globe.pm** for variable definitions.

The **tree_js_2_script.pl** perl program is not very complicated, which is why it is not commented or documented very well (at this point in time. It is heavily dependent on Doxygen and the assumptions and formats in the output tree.js file.

Output

The **tree_js_2_script.pl** perl outputs a script file suitable for the Java TOC Applet.

Chapter 14 html_look_integrate.pl

The **html_look_integrate.pl** perl program and its **html_look_integrate.pm** (name is optional) companion are intended as an internal web spider tool. You give it the starting point (an HTML file) and it traces the hyperlinks. While tracing, it creates a hierarchical table of contents. The operation is complete when there are no more unvisited HTML files referenced from hyperlinks in any of the files.

Overview

This tool came out of the need to provide a table of contents structure to some inherited HTML documentation from an outside vendor which didn't have such. Moreover, the HTML documentation could be updated whenever the vendor released new software or whenever we ran their tools to generate HTML documentation for our enhancements to their software.

The **html_look_integrate.pl** perl program is not perfect, but could be useful. This is how it works.

1. You provide it with one or more top-level starting files.
2. It opens the starting file and looks for any hyperlinks to other files.
3. A list of children files is created for the owning document based on the destination files of the hyperlinks.
   - » If the destination file is a starting file, it is added to the list but its contents are not traced.
   - » Likewise, if the destination file has already been processed through some other path, it is added to the list but its contents are not traced again.
4. For each child file requiring tracing is opened and the step 3 is repeated with the child now being an owning document.

The imperfections of the **html_look_integrate.pl** perl program that immediately come to light:

- If a document is already a type of table of contents with some hierarchical structure, this tracing flattens that structure out giving equal weight to all found hyperlinks.
- If a document makes an off-hand reference hyperlink to a related topic document, this tool may explore that hierarchy early in the process and place its structure out of contents under the "owning" document.
- In terms of tracing, this gives equal weighting to child hyperlink references that are hierarchical and those that mentioned more in passing, such as copyright hyperlinks or cross-reference hyperlinks.
- To avoid loops, this remembers all visited documents and only places them into the hierarchy once.
- This expects all hyperlinks and cross-referencing to be localized. Hyperlinks out into the greater WWW could result in much churning as it builds a tracing structure for the Web instead of for the local HTML documents.

Input

input_scope A file path and name to a perl package used to limit the scope of the search and to help structure the output. The **html_look_integrate.pm** can be used as an example.

html_tmpl template for generated files. This is typically voyant_master_nav.html. Various fields are extracted (inside <head>, top of <body>, bottom of <body>) and inserted into the HTML files.

The **input_scope** file is the key to the successful operation. This file will require tweaking as you experiment with the output.

CAUTION! ***This is a code file, so be careful about syntax.***

The **input_scope** file is required to have:

- **@top_files,** an array of files that determines the starting points for all subsequent traces. If the destination of a hyperlink references one of these array items, further tracing is stopped. The order of the items in this file is important, because it determines the order of items in the traced heirarchy.

- **@ex_as_child,** an array of files that are to be excluded as children and further tracing when found as the destination of a hyperlink reference. The intent of this are to exclude HTML files that might be referenced that are themselves table of content or heirarchy summary files.

You may run the **html_look_integrate.pl** perl many times to tweak the **input_scope** file in order to get the desired output.

Output

The **html_look_integrate.pl** perl generates a `tree.script` file that can be used by the Java TOC Applet.

In addition, this uses the **html_tmpl** template for generated files. This is typically voyant_master_nav.html. Various fields are extracted (inside <head>, top of <body>, bottom of <body>) from the master (**html_tmpl**) and their contents inserted into the HTML files. The purpose of this is to give a consistent look and feel to the HTML system.

In other words, the HTML files are all overwritten. The general content doesn't change. What changes is

- any navigation for browsing at top of <body>.
- any generation date and copyright information at the bottom of <body>.
- any CSS, formats, or applet definitions within the <head>.

Chapter 15 ini_html_gen.pl

The **ini_html_gen.pl** perl program is a special purpose tool for generating documentation for INI files. Although this is a Voyant specific tool for generating HTML documentation for our INI configuration files, those INI files are based on standards defined by others, such as Microsoft and SUN.

Overview

The **ini_html_gen.pl** perl program was created after the log_html_gen.pl tool. Many of the log_html_gen.pl program routines were extracted and placed in globe.pm so that they could be used by both programs.

As with the LOG messages for the log_html_gen.pl tool, the requirements for Technical Publications relating to the INI files were:

- Document the individual INI sections and items.
- Don't add any code bloat to the source INI files.

The INI file syntax is well defined. In general, a section begins with **[some_name]** and ends with an empty **[]**. Inside each section are items, possibly comments, and possibly nested sections. Items have the general syntax:

```
item = value // comment.
```

To aid in readability, items within a section are indented. Nested sections require further levels of indentation.

Because this **ini_html_gen.pl** perl program is based on the log_html_gen.pl perl program, it supports:

- XML tags defined as a comment after an INI item.

- XML tags can reference destination XML tags within the same file or from external files.

A valid argument could be made that self-documented configuration files are useful to the customer. Likewise, a valid argument could be made that self-documented configuration files provide the customer with too much information with which to shoot themselves in the foot. The two camps are still arguing.

The developers have the upper-hand and have tools which auto-generate configuration files without comments.

Hence, because the documentation for an INI file item will most likely reside external from the INI file itself and in order to simplify the XML syntax, the **`ini_html_gen.pl`** perl program takes additional input files that can contain the documentation.

So, descriptions to INI file items can be:

- in-line with the INI item definition itself in the comment section.
- within the INI file somewhere, marked with an XML destination tag.
- in an external file somewhere marked with an XML destination tag.

This approach was chosen, because it puts the documentation much closer to the source (the engineers). Even if a default fully-commented INI file doesn't exist that the engineers maintain, an HTML file checked in near their source code could be maintained. Even failing that, the **`ini_html_gen.pl`** perl program does checks for all exposed INI items to see if documentation exists.

The auto-generation of documentation extract the current syntax of the INI file, create a template HTML document for each INI file item, and then trace the XML tags to locate the plain text descriptions. During the output phase, appropriate entries for the index and table of contents files are generated.

Input

`root_dir` path to place the output files. This is generally something like doc_publish/cref_sysxini/ .

`src_ini_file` is the input source file that has the INI items. This file has a pre-defined format and structure.

html_tmpl template for generated files. This is typically voyant_master_nav.html. The true template is internal to the tool, but this master has information for the <head>, top of <body>, and bottom of <body> which gives a standardize look-and-feel for the whole sub-project that can match the whole project.

org_title [optional] title for the example file. This is plain text that gets exposed in several places in the output.

ext_doc [optional] additional documentation file with XML tag destinations and HTML formatted information. This is useful when the **src_ini_file** does not have any XML tags or documentation, or when programs are based on the same source and have overlaps in the INI configuration.

The **ext_doc** can have documentation for more INI items than are present in the **src_ini_file**.

An example of where this would be useful is when working configuration files expose only those INI items needed for their specific modes of operation or their specific applications. In which case, the input **src_ini_file** could be an application-specific INI file without comments, while **ext_doc** could be a fully-commented and documented master default INI file.

Another example is when you desire documentation for customer-specific INI files that expose their default values.

Output

The goal of the **ini_html_gen.pl** perl program is a series of HTML files that fully document all configuration items exposed in the input INI file.

On the way to that goal, the **ini_html_gen.pl** perl program generates several intermediate files which can be used depending upon the input state of the source INI file.

`INI file with XML tags`: Sometimes the INI file to be documented has no XML tags in the comment area of the configuration item. Hence, you can run this tool to generate an output file with those tags. The output file then becomes an input in a subsequent run of the tool.

Note: Yes, this is convoluted, but it is intended only as a stop-gap measure until engineering is fully on board about how to document INI files. It is hoped that either their fully-documented master default INI file or their tool generated INI files will contain the appropriate XML tags. If they decide not to do that, then this tool can make assumptions about XML tag names, which it does for this temporary output file.

`Text file with documentation and XML destination tags`: If there is a fully-documented master default INI file, this temporary file holds just the XML destination tags and the associated HTML documentation. Later, this file can be used as an additional input file when creating documentation for an application-specific or customer-specific INI file.

Note: It can sometimes be trouble-some to use a valid fully-documented and XML tagged INI file as an additional source for the documentation, because the target XML tags cause unending loops in the tracing process. By extracting the documentation into a separate file, it contains only the destination XML tags. This tool can then be run once to get the documentation from the master and then a second time to create the HTML suite for the specific INI file.

`INI file tagged and documented`: If the source INI file had its documentation initially stored in another file, this creates a self-documented and XML tagged INI file.

Note: This file was used mostly as a test of the tools and proof of concept. This file should be a working INI file that might be cleaner and neater than the original INI file. If Technical Publications was given free reign to help create a self-documenting default master INI file, this particular version could be the one that is handed back after the initial effort and is subsequently maintained.

The various text files that are output were intended to give me a quick leg up in documenting new INI files when the files had no target XML tags, much less destination XML tags with HTML descriptions for the INI items. Hence, during the initial documentation phase, the **`ini_html_gen.pl`** program is sometimes run several times and various aspects of its output are fed back into the tool for yet other runs.

Eventually, though, a documentation file with destination XML tags and its HTML descriptions should be made available for use with subsequent versions of the INI file. As of this point in time until

agreement is reached, it might remain a two-pass process or the tool might be modified into a one-pass process depending on our company decisions.

In any event, the final product is a series of HTML files. Each HTML file has a standard template format which is defined inside of the **`ini_html_gen.pl`** perl program. It leaves the order of the items as they were found in the original INI file. It adds an appropriate overview topic and adds appropriate information for browsing. In addition, an appropriate `tree.script` file is created that can be used by the Java TOC Applet.

The **`ini_html_gen.pl`** perl program has the ability to create an appropriate index file. However, the voyant_nav.pl does a better job. Moreover, the voyant_nav.pl may have to be called at a later point in time with an updated voyant_master_nav.html in order to standardize the look-and-feel.

CAUTION! ***The voyant_nav.pl program does not generate a tree.script file that is the same as intended by the log_html_gen.pl program. Therefore, any shell scripts for creating INI documentation should first backup the tree.script, run voyant_nav.pl, and then overwrite the generated tree.script with the backed up tree.script. Likewise, subsequent shell scripts, such as 50_nav_update.b, should use 55_nav_cp.b on this project.***

Notes:

Chapter 16 log_html_gen.pl

The **log_html_gen.pl** perl program is a Voyant specific tool for generating HTML documentation for our LOG messages.

Unmodified, the **log_html_gen.pl** tools may be of little use to you in your environment, because it relies on the pre-defined format of a company-internal source file.

Overview

Voyant has localized the definition of LOG messages from our product into a small handful of source text files. These files have unique syntax and formatting, which is why an unmodified version of the **log_html_gen.pl** tool may be of little use to you in your environment.

The requirements for Technical Publications were:

- Document the individual LOG messages.
- Don’t add any code bloat to the source files.

The source files have either comment lines or LOG message lines. Each LOG message is on a line of its own with fields delimited by a special character. The engineers left the last field on the line as a free-form field for Technical Publications.

Taking the command regarding code bloat at face value and knowing that sometimes the documentation could not be expected to fit (readably) on one line, we decided to insert XML tags into the source file and the comment fields. The XML tags then could point to XML tags located in external files to find the rest of the documentation.

It turns out, when the engineers said they didn’t want any code bloat as a result of documentation efforts, they really meant that they didn’t want natural groups of LOG messages interrupted with lengthy

documentation blocks. They didn't care about the size of the file. They permitted lengthy documentation blocks at the beginning of the file, at the end of the file, or in between groupings of LOG messages.

Hence, the XML tags in a LOG message comment could point to comment lines which contain destination XML tags and the (lengthy) description of the LOG message.

So, descriptions to LOG messages can be:

- in-line with the LOG message definition itself in the comment section.
- within the LOG message file somewhere, marked with an XML destination tag.
- in an external file somewhere marked with an XML destination tag.

This approach was chosen, because it puts the documentation much closer to the source (the engineers). More importantly, the engineers can change the syntax of the LOG messages at any point in time, which the documentation needs to keep up with. Generating the documentation is the only way to keep up with rapidly changing code (LOG messages).

The auto-generation of documentation extract the current syntax of the LOG message, create a template HTML document for the LOG message, and then trace the XML tags to locate the plain text descriptions. During the output phase, the LOG messages are sorted and appropriate entries for the index and table of contents files are generated.

Many of the key routines of **`log_html_gen.pl`** were generalized further and then extracted into globe.pm, because much of what the ini_html_gen.pl perl program required in terms of handling XML tags was identical.

Input

`root_dir` path to place the output files. This is generally something like doc_publish/cref_sysxini/ .

`message_file` is the input source file that has the LOG messages. This file has a pre-defined format with either comment lines or LOG message lines. The format of the LOG message lines is also pre-defined.

html_tmpl template for generated files. This is typically voyant_master_nav.html. The true template is internal to the tool, but this master has information for the <head>, top of <body>, and bottom of <body> which gives a standardize look-and-feel for the whole sub-project that can match the whole project.

CAUTION! ***Unmodified, the log_html_gen.pl tools may be of little use to you in your environment, because it relies on the pre-defined format of a company-internal source file (message_file).***

Output

The **log_html_gen.pl** perl program generates a series of HTML files based on the individual entries of the LOG message file.

Each HTML file has a standard template format which is defined inside of the **log_html_gen.pl** perl program. It sorts the output alphabetically by LOG message name and adds appropriate information for browsing. In addition, an appropriate tree.script file is created that can be used by the Java TOC Applet.

The **log_html_gen.pl** perl program has the ability to create an appropriate index file. However, the voyant_nav.pl does a better job. Moreover, the voyant_nav.pl may have to be called at a later point in time with an updated voyant_master_nav.html in order to standardize the look-and-feel.

CAUTION! ***The voyant_nav.pl program does not generate a tree.script file that is the same as intended by the log_html_gen.pl program. Therefore, any shell scripts for creating LOG messages should first backup the tree.script, run voyant_nav.pl, and then overwrite the generated tree.script with the backed up tree.script. Likewise, subsequent shell scripts, such as 50_nav_update.b, should use 55_nav_cp.b on this project.***

Notes:

Chapter 17 Input Filters to Doxygen

Doxygen works on C/C++ source files. Occasionally, the file needs to be modified prior to Doxygen viewing it in order to improve its reliability. The Doxygen project files support the specification of input filters.

Input filters are used as part of this project.

- Input filters change comment styles from `//!` into `/**...**/`. The latter are C-style comment blocks that work better with Doxygen, because they group comments together. Individual `//!` do not get picked out of, say, the file header and placed on the first code element.
- Input filters change class definitions into a format that is more standard for Doxygen reporting.
- Input filters change the **`@bug`** Doxygen command into **`@lim`** command that is defined in the project file. The **`@bug`** command outputs "*Bugs and Limitations*"; we prefer seeing "*Limitations and Caveats*".
- Input filters can change IVE program files (Pascal-like programming language) and Perl program files into a format that is more or less recognized by Doxygen.

dox_bug_filter.pl

Input filter for Doxygen for most C/C++ code files. This changes code comments and other items on-the-fly so that it can be more effectively processed by Doxygen.

In particular, the Voyant coding standard calls for `//!` comments. However, these comments are not interpreted as blocks even when placed together. Sometimes comments from a file header were being placed on code items.

By changing the comment style to be C-style, a true comment block could be created with no misinterpretation.

Other changes that this file does include:

- formatting some auto-generated classes to be correct for extraction.
- replacing **@_bug** with **@_lim** so that the word "*bug*" never appears in our output.

dox_ive_filter.pl

Input filter for Doxygen for most IVE code files. This "fakes-out" Doxygen into thinking that IVE (Pascal-like) code files are really C files.

dox_chg_not.pl

Tool for Doxygen for most C/C++ code files. This changes Doxygen commands to a style that is more readable. Forward and backward slashes make my eyes dizzy and aren't in the JavaDoc convention.

This file isn't used any more. Back when we started using Doxygen, we did not have a good handle on how to structure comments, how to format commands, etc. This tool fixes a mistaken direction taken in the early efforts.

dox_comment_chg.pl

Used when Doxygenating C/C++ code files. This is a tool to change the Doxygen comment style from C-style comments **/** ... **/** to comments with **//!**.

The reason is that influential members of the Voyant coding convention group decided that they liked C++ style comments better and did not want to have mixed comment styles in their source files. Hence, our coding conventions specify this style.

This tool changes any existing C-style comments to this **//!** comments. This is used when the technical writer is editing those source files. This change gets checked in.

However, input filters need to be used to change those `//!` comments back into the C-style on-the-fly, because the C-style works better in Doxygen.

pl_comment_change.pl

Changes special comments (##!) inside Perl programs so that they look more C++ like for Doxygen to handle.

It also tries to special case some of the Perl constructs so that they aren't misinterpreted and inappropriately exposed as incorrect C++ constructs.

csh_comment_change.pl

Changes special comments (##!) inside shell scripts so that they look more C++ like for Doxygen to handle.

Notes:

Chapter 18 voyant_master_nav.html

The **voyant_master_nav.html** file is a template file that is used by the voyant_nav.pl program. Its intend is to facilitate standardized look-and-feel, navigation, and copyright information across all HTML files that are in a documentation system.

Overview

The TechPubTools employ standard HTML comments that reside in the HTML file but are not displayed by the browser. The comments are typically used as pairs (**begin** and **end**) and contain a tag name that TechPubTools recognize.

The **voyant_master_nav.html** defines master information for all HTML files in sub-project's directory or even for the entire HTML system. The information residing between the paired comment tags in the master file gets propagated into all HTML files in between their respective comment tags.

The **voyant_master_nav.html** can be used as a template to define your own master file. The name of the file is not important. The 55_nav_gen.b shell script makes calls to the voyant_nav.pl program and passes in whatever master file you specify.

Presently, the **voyant_master_nav.html** defines:

- the cascading style sheet information in addition to hyperlink colors.
- the common navigation that all HTML files should have. This is implemented as "fake-button" through nested HTML tables. This was chosen because at the time, appropriate GIF images for the buttons were not available. It was also believed that the tables would load fast.

- the navigation that Doxygen generated files should have. This uses nested HTML tables for “fake-buttons”.
- the navigation that Mif2Go generated files should have. These are files that come from FrameMaker and are meant to be browsed. This uses nested HTML tables for “fake-buttons”.
- the copyright information that is displayed at the bottom of each topic.
- additional parameters for the Java TOC Applet which allow the topic to be tracked in the table of contents when using the browse buttons (and assuming the book for the topic has already been expanded to at least its first level.)

Minimum Master Definition

Your master files are required to have a certain comment tags defined. Your master file does not have to a functional HTML file, but may be easier to work with and modify if it is completely specified. The comment tags can appear in any order.

<!-- begin voy_header --><!-- end voy_header --> used for definitions such as the cascading style sheet and hyperlink options.

<!-- begin voy_common_top --><!-- end voy_common_top --> used for system-wide common navigation.

<!-- begin voy_fm_book --><!-- end voy_fm_book --> used for navigation that is specific to a FrameMaker book. In particular, this means buttons or hyperlinks for previous and next topics.

<!-- begin voy_dox --><!-- end voy_dox --> used for navigation that is specific to the Doxygen output. This is not specified in the `mif2htm.ini` files, but does appear in files used by Doxygen. Also, the post-processing tools need to know about this.

<!-- begin voy_footer --><!-- end voy_footer --> used for system-wide common footer material. In particular, this can contain the copyright information, date of generation, and document numbers.

<!-- begin voy_html_zap --><!-- end voy_html_zap --> used for defining HTML syntax that is troublesome in the individual files. Typically, this is hard-coded font information that defies control by the cascading style sheet. Hence, each line of information between the **voy_html_zap** tags in the master file contains:

- The opening HTML syntax that you want to remove. This is something like “`<font face`“. This can have incomplete HTML syntax.

- This is the closing syntax to the HTML code that you want to remove. This is usually ">". Everything between that opening syntax and very next occurrence of the closing syntax is removed.
- In addition, sometimes the HTML tag being removed is part of paired tags. This could be something like "". If present, the very next occurrence of this HTML tag after the closing syntax is removed.

<!-- begin voyant_variables --> is a single comment created by Mif2Go. It is used to store information about the FrameMaker topic. This HTML comment contains a **##** separated list of paired items (a flag, **##** separator, and content) needed for navigation.

Note: The tag names in the comment can be changed. However, the change needs to be propagated into `global.pm`, `mif2htm.ini`, the `voyant_master.html` files, and the main header files used by Doxygen (`voyant_head.txt` and `voyant_foot.txt`).

Variables

A small number of variables can be used within your master file to facilitate getting the template to work in HTML files that reside in different directories that could be at different levels.

##rp2start## stands for "relative path to start". This is used to create hyperlinks to a file in the root directory when it isn't known whether the owning HTML file is in the root or in a subdirectory. This is used in the common navigation.

##xgroup## stands for the name of the Doxygen group. This is used in the **voy_dox** navigation and specifies the name of the project.

##xmanual## stands for the name of the current FrameMaker / Mif2Go manual. This is used in the **voy_fm_book** navigation.

##prev## stands for the name of the previous HTML file. This is used in the **voy_fm_book** navigation when previous / next browse buttons are provided to allow the reader to sequential step through the documentation.

##next## stands for the name of the next HTML file. This is used in the **voy_fm_book** navigation when previous / next browse buttons are provided to allow the reader to sequential step through the documentation

Notes:

Chapter 19 TOC Implementation

The index and table of contents need to be flexible to allow sub-projects to be added, deleted, and updated independently of the other projects and the whole system.

In the area of the table of contents, this is accomplished using:

- m_tree.script files
- m_toc.html file
- Java TOC Applet

m_tree.script

The voyant_nav.pl perl program (called from the 50_nav_update.b shell script) generates `tree.script` files on a sub-project basis. These are mini-TOC (table of content) files which are specific to that sub-project.

The format of the `tree.script` files adheres to the guidelines required by the Java TOC Applet.

The voyant_mt_app.pl perl program (called from the 56_nav_script.b shell script) uses these `tree.script` files to generate `m_tree*.script` files which are specific to the whole project or whole HTML system. The top-level `m_tree.script` file nests the individual `m_tree*.script` files from the sub-projects. Together, this creates the master table of contents for the whole system.

Note: *Because the script files for the table of contents are generated and exist independently, the entire HTML system becomes more modular and easier to maintain.*

m_toc.html

The existence of the script files does not mean that they or their associated Java TOC Applet for the table of contents are accessible in the HTML system.

The hooks to plug the Java TOC Applet into the project and the HTML system are in the m_toc.html file. (Topic tracking in the TOC is accomplished by some parameters for the Java TOC Applet located in the voyant_master_nav.html file.)

In the TechPubTools implementation, the **`_start_here.html`** creates a frameset and displays the **`m_toc.html`** in the left-hand "navigation" frame.

The **`m_toc.html`** file defines how the Java table of contents (TOC) applet plugs into the system. It provides the input parameters to the applet, such as the size of the applet and the initial script (**`m_tree.script`**) that is displayed as the table of contents. In addition, the **`m_toc.html`** file can contain information above and below the table of contents, such as navigation elements and overview topics.

More details about the Java TOC Applet parameters are discussed only in its documentation, which must be purchased.

Logical Extensions of the Applet

Although the Java TOC Applet is used for the table of contents, it could theoretically be used for the index, as well.

I explored this direction for a short time with the voyant_indexer.pl perl program. It still outputs **`m_idx.script`** which can be plugged into the Java TOC Applet (such as on a **`m_idx.html`** file) in the same manner that the m_tree.script files are plugged into the m_toc.html file.

I stopped pursuing this because:

1. There are loading delays when the applet first reads in the script file. The delays in going back and forth between the table of contents and index would be too great, particularly for very large indices.
2. Some of my projects create large indices on the order of 20,000 entries or more, depending upon whether or not word-chunking is turned on. It may have necessitated breaking the index into nested script files by letter, much like the table of contents, which introduces further loading delays.
3. The script files cannot be searched from the browser. Therefore, it can be more difficult to locate entries in a large index even after expanding it.
4. The window for the applet (even if put into a separate browser instance) place limitations on the display region and scrolling.
5. Most important of all, the present design of multiple HTML files for the index works; the pages load fast, scroll easy, and can be searched.

Still, there are other areas and other information slicing-and-dicing that this Java TOC Applet can be used for.

Chapter 20 voyant_master_index.html

The **voyant_master_index.html** file is used by the voyant_indexer.pl program to create a series of tree files that represent the table of contents navigation of the HTML system.

More so than the **voyant_master_nav.html** file, the **voyant_master_index.html** file is intended to be used over and over as a template.

The TechPubTools employ standard HTML comments that reside in the HTML file but are not displayed by the browser. The comments are used as pairs (**begin** and **end**) and contain a tag name that TechPubTools recognize.

The **voyant_master_index.html** can be used as a template to define your own master file. The name of the file is not important. The 56_nav_index.b shell script makes calls to the voyant_indexer.pl program and passes in whatever master file you specify.

Presently, the **voyant_master_index.html** defines:

- the cascading style sheet information in addition to hyperlink colors.
- the outline of the structure to be used for all navigation pages.
- the links to other pages of the navigation. This can include fake-buttons at the top and bottom to the Contents, Index, and Full-Text Search.

Together with the **_start_here.html** file, a dual pane navigation system is created. The series of **m_idx** files that are generated based on this template file are meant to be displayed in the **target=treefrm**. The contents that they control are meant to be displayed in the **target=basefrm**.

Your master files are required to have a certain comment tags defined. Your master file has to be a functional HTML file with the proper **<html>**, **<head>**, and **<body>** paired tags.

<!-- begin voy_header --><!-- end voy_header --> used for definitions such as the cascading style sheet and hyperlink options.

<!-- begin voy_structure --><!-- end voy_structure --> used for defining the exact location where generated table of contents text is to be placed. You can place hyperlinks to other navigation control outside of these two comment tags. An example of that would be “fake-buttons” to the files associated with the letters of the alphabet.

Note: The tag names in the comment can be changed. However, the change needs to be propagated into `global.pm`, `mif2htm.ini`, the `voyant_master.html` files, and the main header files used by Doxygen (`voyant_head.txt` and `voyant_foot.txt`).

Chapter 21 Common Files

Several files generally copied into the directories before Doxygen or Mif2Go is run. Although laziness is part of the reason they are copied instead of referenced to a single location, having them in those directories facilitates testing and makes each directory a stand-alone system.

Default HTML Files for Doxygen

Doxygen only generates the files that it needs for the system. However, in order to create a comprehensive system that spans Doxygen projects, consistent navigation is required.

Hence these default files are copied by the shell scripts into the Doxygen code reference directories before Doxygen is run. Doxygen overwrites these files if appropriate.

- `files.html`
- `classes.html`
- `annotated.html`
- `functions.html`
- `globals.html`
- `hierarchy.html`
- `modules.html`

Navigation GIF files

The following files are used in the generated `tree.html` files and in the `m_tree_` navigation files. To facilitate operation, they are copied by the shell scripts into the directories. These files could be referenced to a single location, but copying them makes each directory a complete system in and of itself.

- `clear.gif`
- `voyant.gif`
- `nav_folderopen.gif`
- `nav_folderclosed.gif`
- `nav_doc.gif`
- `voyant-mdblu.gif`

Cascading Style Sheet

In order to provide a consistent look and feel, the `voyant_fm.css` cascading style sheet is employed. Although it could be referenced in a specific location, laziness in implementation and testing has the shell script copy this into the directories of the project. Copying this file makes each directory a complete system in and of itself.

- Part of the `voyant_fm.css` was created by Mif2Go with the first manual that it exported. Voyant's Technical Publications Department has templates that its documents consistently follow.
- Part of the `voyant_fm.css` was copied from `doxygen.css`, which was created by Doxygen.
- A final part of the `voyant_fm.css` was defined during the development of the Perl tools that generate the table of contents and index files.

The resulting `voyant_fm.css` was then modified and streamlined to tailor it for our needs.

Index