SDG:System Documentation Guidelines	
Version:	0.0.0
Date:	August 24, 2007
Purpose:	Organization and content of P545 System Documentation. This document serves as an example, implementing the requirements in Latex
Status:	in progress
Author:	Steven D. Johnson

System Documentation Guidelines

Contents

Technical Profile

These guidelines outline the structure, purpose, and format of system documentation for P545 projects. A "system" an entire system, a subsystem of something larger, a component—module, object, class, library, and API, or even a single routine. This document also serves as an example, describing a *Latex* implementation for formatting system documentation in Sections 2–5.

1 Requirements

1.1 Structure and Purpose of System Documentation

SPECIFICATION 1.1 *Section level organization of system documentation is given in Figure 1.*

The remainder of this section discusses the purpose and content of each section. Generally, each section develops a successively more concrete view of the target realization. In general, specifics are deferred to later sections insofar as it makes sense. For instance, the *requirements* should not prescribe design decisions; and the *design* should not prescribe representation details. However, there are no precise "boundaries" for what is specified where. It depends on the nature of the component being described. these sections, and they are all describing the same thing at differing abstraction levels.

1.1.1 Requirements

The Requirements section specifies *what* the component under design does in terms of its *externally observable* behavior. Requirement specifications may include such properties as:

- *Functionality,* the input-output relations.
- *Preconditions* or "assumptions," are conditions for correct use.
- *Postconditions* or "guarantees," including not only output values but also such things as effects on call-by-reference argements, file space, etc.
- *Invariants*, such as safety and liveness conditions that are preserved by the executing component.

1. *Requirements.* What the system does.
2. *Design.* How the requirements are satisfied.
3. *Implementation.* Key representations, algorithms, etc.
4. *Coding.* Indexed presentation of source code.
5. *Testing.* Purpose and procedures.

A. *Developer Instructions.* `make` procedures, etc.

B. *User Instructions.* End-user procedures.

Figure 1: Organization of System Documentation

- *Constraints* on resources such as time, space, etc.
- *Validation.* The requirements may include a collection of specific observable (i.e. input/output) behaviors to which the delivered realization must comply. These may be thought of as being provided by the End User (or customer) for determining minimal satisfactory functionality.

The "formal" requirements statement consists of a numbered sequence of individual *requirement specifications*, like Requirement 1.1 at the beginning of Section 1. In critical applications, the subsequent sections are expected to address each of these individually.

1.1.2 Design

The *Design* section explains *how* the requirements are satisfied. For this reason, it is usually organized according to component functionality (rather than architecture, as is the case in the *Implementation* section).

The design is presented abstractly, and routine representation details are deferred. It typically includes, for example, graph depictions of data structures or control-flow. Key algorithms may be presented in abbreviated form (e.g. pseudo-code) the main goal being to show mathematically how requirements are addressed.

Ideally, the design description gives just enough information for someone with sufficient programming expertise to develop an equivalent implementation on their own.

Design *verification* is a comparison of two levels of *description* (as opposed to *testing* the actual realization, see Section 1.1.5). In critical applications, it is necessary to explain how each requirement specification is satisfied by the design. The means of verification ranges from demonstration by model simulation to machine-checked mathematical proof, although in commone practice may be merely a careful, more-or-less rigorous English explanation.

1.1.3 Implementation

The *Implementation* section presents "key" representation details, including the overall architecture of the component. It should not include incidental coding details that can be readily understood by inspection of the source code. Instead, this section gives the "lay of the land," that a competent programmer would need to know before delving into the low-level coding details.

Specific design specification statements, if any, should be addressed. This is another level at which the term "*verification*" is applicable.

1.1.4 Code

Source code is processed by a *code documentation* tool. *Doxygen*[1] that generates navigation indices, such as call graphs. These tools often have comment formatting provisions as well, allowing source-comments to be integrated logically in the higher-level system documentation.

However, the primary purpose of source-comments to providing *local* guidance in the immediate code context. Hence, these comments are generally insufficient for the higher purpose of the *Implementation* section.

1.1.5 Test

In contrast to *validation* (Sec. 1.1.1) and *verification* (Sec. 1.1.2), *testing* refers to execution of the component realization (hardware device, object code, etc.) against a selected sample of inputs and expected outputs.

The test of interest in this section do not include routine tracing for the purpose of programming, but rather, a cumulative suite of fixed tests whose purposes include final validation against end-user requirements, and *regression* testing against revisions, diagnoses of failures in the field, and so forth.

These tests should be automated for repeatability, and anomales in testing must be tracked and resolved prior to release of a component. This section includes *both* test specifications *and* documentation of repeatable test procedures.

1.2 Tagged Specifications

SPECIFICATION 1.2 *Formal specification statements are identified by locator tags indicating the section in which they appear and a sequence number.*

2 Design

2.1 LaTeXImplementation

Support for system documentation as specified in Section 1 is provided in a collection of macros [name] incorporated in the document prelude. The SDG prelude loads packages makeidx, **hyperref**, and url to enable hyper-text linking. Package moreverb is used for in-lining files and other similar purposes.

End users can use `\label`, `\ref`, and `\pageref` in the usual way to incorporate hyper-text links, provided the document is generated with `pdflatex`, or its equivalent.

[1] `http://www.stack.nl/~dimitri/doxygen/index.html` is used in P545 unless it is superceded by an equivalent tool provided by the project development environment.

```
\documentclass{article}
\input{../SDG-prelude.tex}

\begin{document}

\DocHeader{TAG}{TITLE}{VERSION}{DATE}{PURPOSE}{STATUS}{AUTHOR}

\begin{TechnicalProfile} ... \end{TechnicalProfile}
\begin{Requirements} ... \end{Requirements}
\begin{Design} ... \end{Design}
\begin{Implementation} ... \end{Implementation}
\begin{Code} ... \end{Code}
\begin{Test} ... \end{Test}

\appendix

\SpecDigest

\end{document}
```

Figure 2: SDG top-level template

2.1.1 Document Organization

SPECIFICATION 2.1 *Latex environments* `Requirements`, `Design`, `Implementation`, `Code`, *and* `Test` *do the sectioning according to Specificication 1.1 (See Fig. 2). They include provisions for inserting a digest of specification statements at the end of each section.*

Each of the sectioning environments creates and initializes an output file—`Rspec.tex`, `Dspec.tex`, `Ispec.tex`, `Cspec.tex`, or `Tspec.tex`, depending on which section is in effect. These files are used to accumulate specification statements (See `\Spec` for inclusion later in a specification digest (See `\SpecDigest`).

2.1.2 Tagged Specifications

SPECIFICATION 2.2 *A specification statement is generated by the form* `\Spec`{*name*} ... `\EndSpec`. *A tag has the form §.n, where § is the section number and n is a sequence number [Requirement* **??***].*

Argument *name* is used to label the statement via `\label`, so that `\ref`{*name*} generates a reference to the tag, and `\pageref`{*name*} generates the page number on which the specification statement appears. The hyperref package generates hyper-text links in PDF target documents.

3 Implementation

3.1 LaTeX

3.1.1 Packages

The following packages are used:

1. makeidx generates an index according to `\index` commands.
2. hyperref generates hyper-text links for `\ref`{*label*} and `\pageref`{*label*} commands.
3. url generates links for `\url`{*http* : // . . .}

3.1.2 Document Information Header

The macro

`\DocInfo`{*tag*}{*title*}{*version*}{*date*}{*purpose*}{*status*}{*author*}

generates a header table identifying the document:

tag:*title*	
VERSION:	*version*
DATE:	*date*
PURPOSE:	*purpose*
STATUS:	*status*
AUTHOR:	*author*

Argument *tag* is for external use in identifying the document.

3.1.3 Document Organization [§2.1.1]

Sectioning nvironments `Requirements`, `Design`, etc. [Spec. 2.1], invoke instances of the

`SDGsection`{*name*}{*file*}

environment, which sets things up sectioning. It initializes an output *file* for accumulating table entries generated by `\Spec` commands (Sec. **??**, below). The resulting table is read into the document at the close of the section to generate a digest of specification statements.

3.1.4 Tagged Specifications

The command

`\Spec`{*name*} *statement* `\EndSpec`

generates specification statements and caches them for later reproduction in the Specification Digest table.[2] Argument *name* is used to `\label` the statement. The tag associated with the label is generated by a `\newtheorem` environment called `SpecStmt` keyed to the section number.

`\Spec` puts LaTeX into verbatim mode which reads the ensuing *statement* up to the endmarker, `\EndSpec`. This "string" is copied to two files:

1. a cache file for immediate use. The cache copy is re-read immediately using `\input`, which evaluates and formats it.

2. a file for accumulating a table of all the specification statements in the section. `\Spec` appends a line of the form

```
\ref{name} & statement \cr\hline
```

[2] `\Spec` is *almost but not quite* and environment. It should be one.

4 Code

The SDG macro file is called `SDT-prelude.tex`.

5 Test

SPECIFICATION 5.1 *The source file for this document,* `SDG.tex`*, exercises all the macros in* `SDG-prelude.tex`*. The command* `make sdg` *the formatted hypertext document* `SDG.pdf`

References

1. Leslie Lamport. *Latex*. ...

2. .... *The Latex Companion*. ...

A Specification Digest

Requirements	
1.1	Section level organization of system documentation is given in Figure 1.
1.2	Formal specification statements are identified by locator tags indicating the section in which they appear and a sequence number.

Design	
2.1	Latex environments `Requirements`, `Design`, `Implementation`, `Code`, and `Test` do the sectioning according to Specificication 1.1 (See Fig. 2). They include provisions for inserting a digest of specification statements at the end of each section.
2.2	A specification statement is generated by the form `\Spec`{*name*} ... `\EndSpec`. A tag has the form §.*n*, where § is the section number and *n* is a sequence number [Requirement **??**].

Implementation

Code

Test	
5.1	The source file for this document, `SDG.tex`, exercises all the macros in `SDG-prelude.tex`. The command `make sdg` the formatted hypertext document `SDG.pdf`

B Developer Instructions

See the `Makefile`. Debugging and enhancement are complicated by the fact that some LaTeXerrors may be propagated in the output files (e.g. `Rspec.tex`) and will therefore persist across multiple calls to latex. Because of this it should be a routine part of development to remove the `.aux` files before re-running the formatter.

C User Instructions

The skeleton template for generating documentation using the SDG macros is shown below

```
\documentclass{article}
\input{../SDG-prelude.tex}

\begin{document}

\DocHeader{TAG}{TITLE}{VERSION}{DATE}{PURPOSE}{STATUS}{AUTHOR}

\begin{TechnicalProfile} ... \end{TechnicalProfile}
\begin{Requirements} ... \end{Requirements}
\begin{Design} ... \end{Design}
\begin{Implementation} ... \end{Implementation}
\begin{Code} ... \end{Code}
\begin{Test} ... \end{Test}

\appendix

\SpecDigest

\end{document}
```

C.1 Using SDG

To generate a document from *file*.tex invoke

```
rm file.aux
pdflatex file
makeindex file
pdflatex file
pdflatex file
```

Note: The SDG-prelude.tex macros are rather sensitive to formatting errors, whose effects often carry over repeated pdflatex invokations. Because of this, it is a wise habit to *remove the* file.aux file between corrections.

Repeated invokations of pdflatex are needed to resolve internal cross-references. The resulting document is named *file*.pdf.

C.2 Sectioning

\begin—tt{*name*} ... \end{*name*}

The Latex environments TechnicalProfile, Requirements, Design, Implementation, Code, and Test must appear in that order, although this isn't checked. They

set up the bookkeeping to generate an index of specification statements (See the `\Spec` command in Section **??**, but otherwise are equivalent to `\section`{*name*} commands.

The `\appendix` command and subsequent `\section`s are not required; they illustrate how to append supplementary documentation.

C.3 Document Information Header

`\DocInfo`{*tag*}{*title*}{*version*}{*date*}{*purpose*}{*status*}{*author*}

`\DocInfo` generates a table of the form.

tag:*title*	
VERSION:	*version*
DATE:	*date*
PURPOSE:	*purpose*
STATUS:	*status*
AUTHOR:	*author(s)*

tag is a prefix that may be used externally to identify this document. So it should be a short string. The remaining arguments may be any length, but must be a single paragraph; *purpose*, in particulasr, should be a short, description of the subject of the documentation.

C.4 Document Header

`\DocHeader`{*tag*}{*title*}{*version*}{*date*}{*purpose*}{*status*}{*author*}

`\DocHeader` generates a `\DocInfo` table and adds a title line and table of contents. Its arguments are passed directly to `\DocInfo`.

C.5 Specification Tagging

`\Spec`{*label*} *statement* `\EndSpec`

`\Spec` is an `environment`-like command in that its *statement* argument need not be enclosed in braces but is instead terminated by the keyword `\EndSpec` (Note: not `\end{Spec}`). *statement* can be any single paragraph. It is displayed in place, NS *statement* is also recorded in one of the files `Rspec.tex`, `Dspec.tex`, `Ispec.tex`, `Cspec.tex`, or `Tspec.tex`, depending on which section is in effect, for re-use as an index elsewhere in the document (see `\SpecDigest`. Do not use `\Spec` outside these predetermined sections.

label is an identifier used to label the statement for use in cross-referencing `\ref` and `\pageref` commands.

EXAMPLE

```
\Spec{rqmt:tags}
Formal specification statements are identified
by locator\index{locator} tags\index{tag} indicating
the section in which they appear and a unique
sequence number.
\EndSpec
```

results in the formatted statement

> SPECIFICATION **1.2** Formal specification statements are identified by locator tags indicating the section in which they appear and a unique sequence number.

The locator **1.2** is generated by `\Spec` and may be referenced by `\ref{rqmt:tags}`. The first number indicates the section number, and the second is a sequence number.

Index