sphinxcontrib-specdomain Documentation

Release 1.04.02

BCDA, Advanced Photon Source, Argonne National Laboratory

March 26, 2016

Contents

This Python package is an extension for Sphinx (http://sphinx.pocoo.org/), to document SPEC macro source code files. SPEC (http://www.certif.com) is control software for X-Ray Diffraction and Data Acquisition of scientific instruments.

This code extracts useful documentation from SPEC macro source code files.

home http://specdomain.readthedocs.org

git https://github.com/prjemian/specdomain

PyPI https://pypi.python.org/pypi/sphinxcontrib-specdomain

CHAPTER 1

Contents

1.1 How to use Sphinx to Document SPEC Macro Source Code Files

1.1.1 How to Download and Install the SPEC support into Sphinx

use package manager

Use one of these methods to get specdomain from pypi.python.org:

```
pip install -U sphinxcontrib-specdomain
```

or:

```
easy_install -U sphinxcontrib-specdomain
```

install from source

1. download from the subversion repository
2. install into Python
3. test the installation

Requires [1]

- Python 2.7 or greater
- Sphinx 1.1.1 or greater

Download

Retrieve the support package from our subversion repository:

```
svn co https://subversion.xray.aps.anl.gov/bcdaext/specdomain/trunk/src/specdomain/ /tmp/specdomain
```

[1] The developer used Python 2.7.2 and Sphinx 1.1.2 while writing this support. Older versions may work but have not been tested.

Install

Continuing from the download above, use the setup tools to install the package somewhere on your PYTHONPATH (you may need admin rights to install into your Python). This command shows how to install into Python's *site-packages* directory:

```
cd /tmp/specdomain
python setup.py install
```

1.1.2 How to Create and Configure a Sphinx project to Document SPEC Macros

Tip: for more information on this topic, refer to the SPhinx tutorial: http://sphinx.pocoo.org/tutorial.html

Decide which configuration

In-source configuration or **Out-of-source** configuration?

Tip: Most likely, you will want to keep the Sphinx documentation files in a separate directory from the SPEC macros. Then again, maybe not.

In-source configuration

An *in-source* configuration is where the Sphinx `.rst` files are in **the same directory** as the SPEC macro files.

Here is a graphical example:

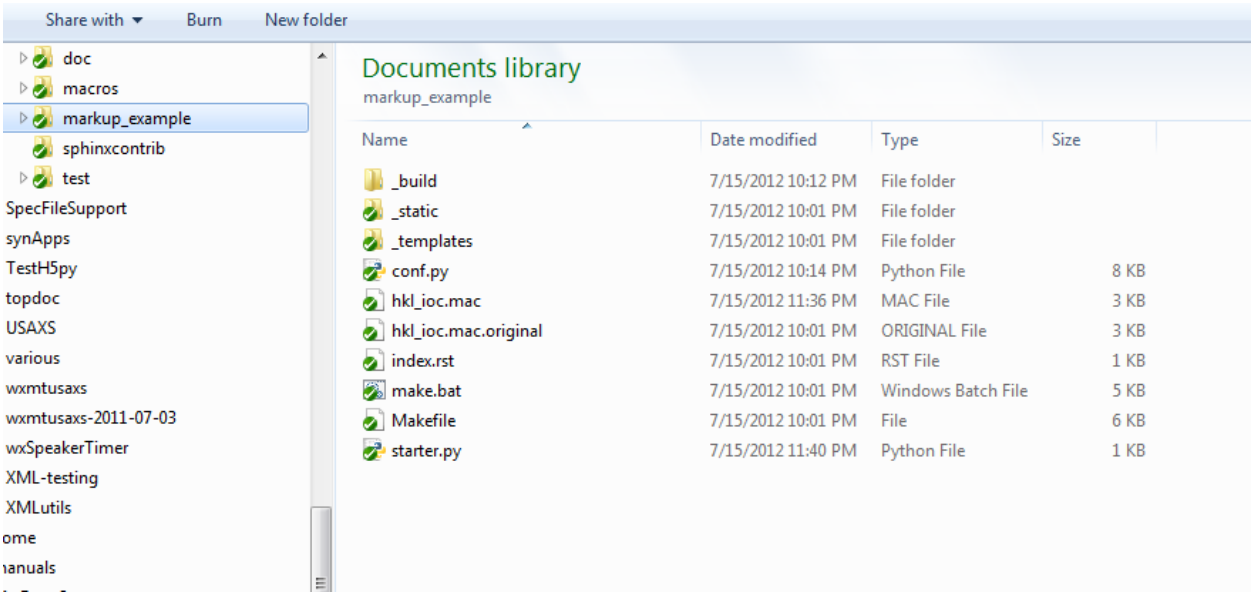

Fig. 1.1: Example directory of an in-source configuration

Out-of-source configuration

An *out-of-source* configuration is where the Sphinx `.rst` files are in **a separate directory** from the SPEC macro files.

Here is a graphical example: [1]

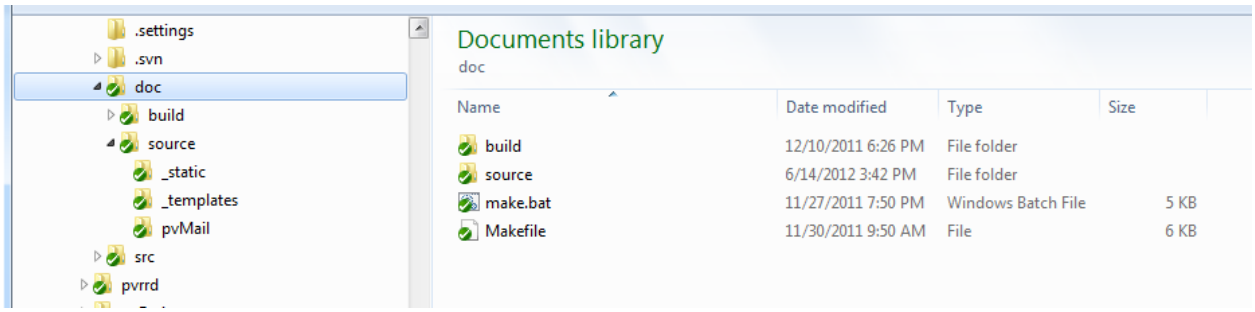

Fig. 1.2: Example build directory of an out-of-source configuration

The source files are located in the *source* directory (child of the build directory):

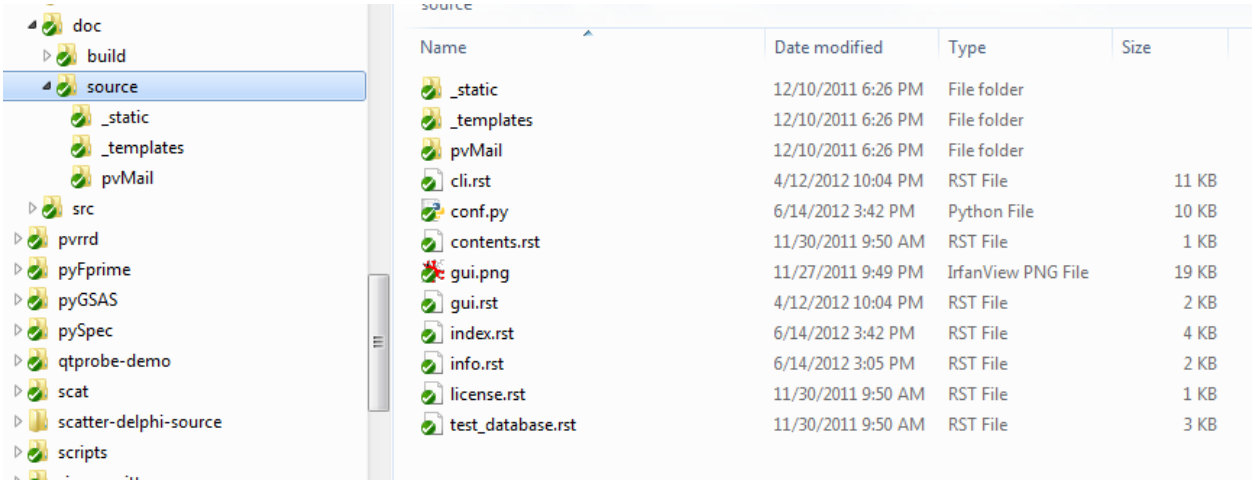

Fig. 1.3: Example *source* directory of an out-of-source configuration

Caution: FIXME: these are Python files and project directories. The figure *should* be shown with SPEC macro files out-of-source.

Create the Sphinx documentation tree

Testing the installation

These instructions are written to help you test if you have installed *specdomain* correctly. They use an *in-source* configuration.

[1] The green check boxes correspond to the status of each item in the version control system.

Make a new sandbox directory to try this out:

```
mkdir /tmp/sandbox
cd /tmp/sandbox
```

Create a Sphinx configuration in this directory by running:

```
sphinx-quickstart
```

Here's the full session:

```
    jemian@como-ubuntu64:/tmp/sandbox$ sphinx-quickstart
    Welcome to the Sphinx 1.1.2 quickstart utility.

    Please enter values for the following settings (just press Enter to
    accept a default value, if one is given in brackets).

    Enter the root path for documentation.
    > Root path for the documentation [.]:

    You have two options for placing the build directory for Sphinx output.
    Either, you use a directory "_build" within the root path, or you separate
    "source" and "build" directories within the root path.
    > Separate source and build directories (y/N) [n]:

    Inside the root directory, two more directories will be created; "_templates"
    for custom HTML templates and "_static" for custom stylesheets and other static
    files. You can enter another prefix (such as ".") to replace the underscore.
    > Name prefix for templates and static dir [_]:

    The project name will occur in several places in the built documentation.
    > Project name: sandbox
    > Author name(s): sandy

    Sphinx has the notion of a "version" and a "release" for the
    software. Each version can have multiple releases. For example, for
    Python the version is something like 2.5 or 3.0, while the release is
    something like 2.5.1 or 3.0a1.  If you don't need this dual structure,
    just set both to the same value.
    > Project version: test
    > Project release [test]:

    The file name suffix for source files. Commonly, this is either ".txt"
    or ".rst".  Only files with this suffix are considered documents.
    > Source file suffix [.rst]:

    One document is special in that it is considered the top node of the
    "contents tree", that is, it is the root of the hierarchical structure
    of the documents. Normally, this is "index", but if your "index"
    document is a custom template, you can also set this to another filename.
    > Name of your master document (without suffix) [index]:

    Sphinx can also add configuration for epub output:
    > Do you want to use the epub builder (y/N) [n]:

    Please indicate if you want to use one of the following Sphinx extensions:
    > autodoc: automatically insert docstrings from modules (y/N) [n]: y
    > doctest: automatically test code snippets in doctest blocks (y/N) [n]:
    > intersphinx: link between Sphinx documentation of different projects (y/N) [n]:
    > todo: write "todo" entries that can be shown or hidden on build (y/N) [n]:
```

```
    > coverage: checks for documentation coverage (y/N) [n]:
    > pngmath: include math, rendered as PNG images (y/N) [n]:
    > mathjax: include math, rendered in the browser by MathJax (y/N) [n]:
    > ifconfig: conditional inclusion of content based on config values (y/N) [n]:
    > viewcode: include links to the source code of documented Python objects (y/N) [n]: y

    A Makefile and a Windows command file can be generated for you so that you
    only have to run e.g. `make html' instead of invoking sphinx-build
    directly.
    > Create Makefile? (Y/n) [y]:
    > Create Windows command file? (Y/n) [y]:

    Creating file ./conf.py.
    Creating file ./index.rst.
    Creating file ./Makefile.
    Creating file ./make.bat.

    Finished: An initial directory structure has been created.

    You should now populate your master file ./index.rst and create other documentation
    source files. Use the Makefile to build the docs, like so:
       make builder
    where "builder" is one of the supported builders, e.g. html, latex or linkcheck.

    jemian@como-ubuntu64:/tmp/sandbox$
```

In case you missed them, these are the non-default answers supplied:

prompt	response
`> Project name:`	*sandbox*
`> Author name(s):`	*sandy*
`> Project version:`	*test*
`> autodoc: automatically insert docstrings from modules (y/N) [n]:`	*y*
`> viewcode: include links to the source code of documented Python objects (y/N) [n]:`	*y*

Configure: Changes to `conf.py`

Edit the new file *conf.py* and add these two lines to the extensions list after line 28:

```
# this says ${PYTHONPATH)/sphinxcontrib/specdomain.py must be found
extensions.append('sphinxcontrib.specdomain')
```

If you wish, you can also change the *html_theme* from the *default* to *sphinxdoc* or *agogo* or one of the others. Check the Sphinx documentation for the choices. To change the theme, look on line 97 (or thereabouts) and change:

```
html_theme = 'default'
```

to:

```
html_theme = 'sphinxdoc'
```

1.1.3 How to Build the Documentation from a Sphinx project

Briefly:

```
make html
```

Load the file `index.html` into your browser and examine the results. The file path is different depending if you have an *in-source* or *out-of-source* configuration.

***in-source* path:** `_build/html/index.html`

***out-of-source* path:** `build/html/index.html`

Test

For our testing purposes, we'll document the *aalength.mac* macro file from the *Install* section. Edit the new file *index.rst* and add this line at line 14. Make sure it lines up at the left in column 1:

```
.. autospecmacro:: ../specdomain/doc/aalength.mac
```

Build the HTML documentation:

```
make html
```

View the documentation using a web browser such as *firefox*:

```
firefox _build/html/index.html &
```

You should see a page that looks like this, if nothing went wrong.

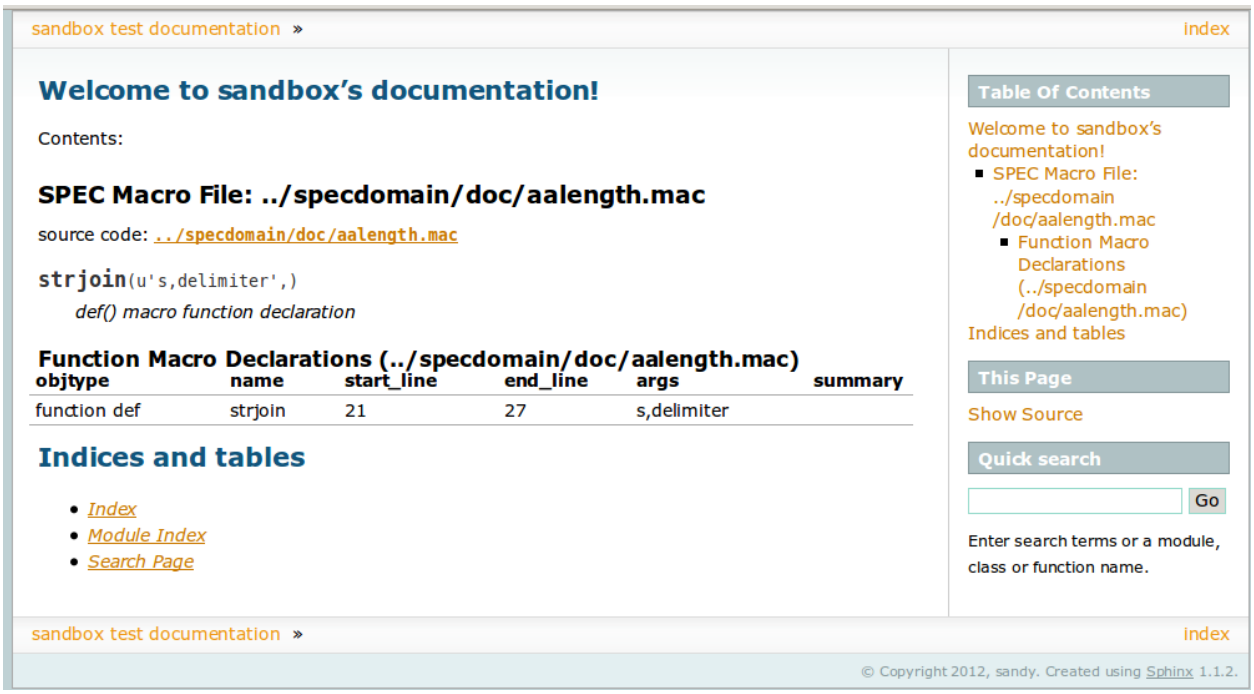

Fig. 1.4: Documentation of the **aalength.mac** file.

1.1.4 How to Markup a SPEC Macro File

1. start with SPEC macro file that is not marked up
2. create a Sphinx documentation project
3. apply markup
4. test

Tip: In addition to the Sphinx documentation (http://sphinx.pocoo.org), a good reference how to document your macros can be found here: http://stackoverflow.com/questions/4547849/good-examples-of-python-docstrings-for-sphinx but a Google search for *sphinx python docstrings examples* will turn up a wealth of alternatives.

Basic SPEC Macro file

This example is a SPEC macro file from the APS subversion repository: https://subversion.xray.aps.anl.gov/spec/macros/trunk/common/hkl_ioc.mac because it is simple, brief, does not contain references to other macro files, provides its documentation in SPEC comments, and has not been marked up previously for documentation with Sphinx. Here is the file *hkl_ioc.mac* in its entirety.

```
#############################################################################
#**************SPEC macros for the Advanced Photon Source**********************
#############################################################################
#
# Beamline/Sector: 4ID
#
# Macro Package: hkl_ioc.mac
#
# Version: 1.0 (August,2005)
#
# Description: A user defined calcHKL to write the current HKL postion to a
#              soft IOC. It requires spec softioc running.
#
# Written by: X. Jiao 08/08/2005
#
# Modified by:
#
# User macros: ioc_HKL -> to turn on/off the feature of putting HKL to shared
#                         memory.
#
# Internal macros: ioc_put_HKL -> write HKL to the soft IOC
#
# Modification history:
#
#$Log$
#Revision 1.3  2006/05/22 20:34:35  jiaox
#removed unsed lines in ioc_HKL.
#
#Revision 1.2  2006/05/11 17:46:31  jiaox
#Added CVS Log entry.
#
```

```
#=============================================================================

#=============================================================================
# preload check/setting here
#=============================================================================

cdef("user_save_HKL","","ioc_HKL")
cdef("calcHKL","calc(2); user_save_HKL;","ioc_HKL")

#=============================================================================
# global variables defined here
#=============================================================================
if( unset("SIOC_PV") ) {
global SIOC_PV
local foo tmp[]
SIOC_PREFIX="4id"
unix("hostname | cut -f1 -d.",foo)
split(foo,tmp,"\n")
foo=tmp[0]
SIOC_PV = sprintf("%s:%s:spec",SIOC_PREFIX,foo)
printf("Spec soft IOC PV(SIOC_PV): %s",SIOC_PV)
}

#=============================================================================
# user macros defined here
#=============================================================================
def ioc_HKL '{

    if($# != 1) { eprint "Usage: ioc_HKL on/off ";exit}


    if(("$1" == "on")) {
       cdef("user_save_HKL","ioc_put_HKL","ioc_HKL","0x20")
       print "Now put HKL to softioc."
       exit
    }
    if(("$1" == "off")) {
       cdef("user_save_HKL","","ioc_HKL","delete")
       print "Stop put HKL to softioc."
       exit
    }
    eprint "Usage: ioc_HKL on/off "
}'


#=============================================================================
# internal macros defined here
#=============================================================================
def ioc_put_HKL '
    epics_put(sprintf("%s:H",SIOC_PV),H)
    epics_put(sprintf("%s:K",SIOC_PV),K)
    epics_put(sprintf("%s:L",SIOC_PV),L)
    epics_put(sprintf("%s:NPTS",SIOC_PV),NPTS)
'
```

Create a Sphinx Project

Tip: Use an *In-source configuration*

Make a project directory and change directory into it. (On my development system, this directory is called `../markup_example`.) Copy the file above into this directory with the name *hkl_ioc.mac*. Then run:

```
sphinx-quickstart
```

Take most of the defaults. These are the non-defaults for the example project:

prompt	response
`> Project name:`	*example*
`> Author name(s):`	*SPEC Macro Writer*
`> Project version:`	*1*
`> autodoc: automatically insert docstrings from modules (y/N) [n]:`	*y*
`> viewcode: include links to the source code of documented Python objects (y/N) [n]:`	*y*

Edit the document `index.rst` so it looks like this:

```
.. example documentation master file, created by
   sphinx-quickstart on Sun Jul 15 17:54:46 2012.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to example's documentation!
===================================

.. autospecmacro:: hkl_ioc.mac

Contents:

.. toctree::
   :maxdepth: 2



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Edit *conf.py* and make these changes:

change	directions
`sys.path.insert(0, os.path.abspath('..'))`	*replace* line 19
`extensions.append( 'sphinxcontrib.specdomain' )`	insert *after* line 28

The revised file should look like this file: `../markup_example/conf.py`.

Now, build the documentation by typing:

```
make html
firefox _build/html/index.html &
```

You should expect a page that looks like this figure:

Apply Markup

Tip: We will edit the *hkl_ioc.mac* file now. Before we start applying markup, it's a good idea to make a backup copy of the original:

```
cp hkl_ioc.mac  hkl_ioc.mac.original
```

Here are the steps to consider when converting SPEC comments to reST markup.

1. identify the SPEC comments
2. extended comments (docstrings) #. global docstring #. macro docstring #. others are ignored by Sphinx
3. descriptive comments

SPEC comments

Obvious as this sounds, it may help someone to see that the SPEC comments are on lines 1-32. It is easy to place this entire block within an extended comment [1] (known hereafter as a *docstring*). Make these additions to lines 1 and 32:

line	new
1	"""#==========
32	#=========="""

But this is not enough, as the content will look like a wreck.

```
#============================================================================
#**************SPEC macros for the Advanced Photon Source**********************
#============================================================================
# # Beamline/Sector: 4ID # # Macro Package: hkl_ioc.mac # # Version: 1.0 (August,2005) #
```

... and so forth

So, there is a choice to make. Either:

1. format the comment as literal text (least work)
2. reformat the as reST. (more work, looks better)

Literal text To format a SPEC comment as literal text, consider this brief SPEC comment:

[1] a SPEC *extended comment* is text that is surrounded by three double-quote characters (""""), such as:

""""this triple-quoted text is a docstring""""

In the Python language, this is known as a docstring. But, **unlike Python**, SPEC does not recognize single quotes to mark extended comments. Only use the double quote character.

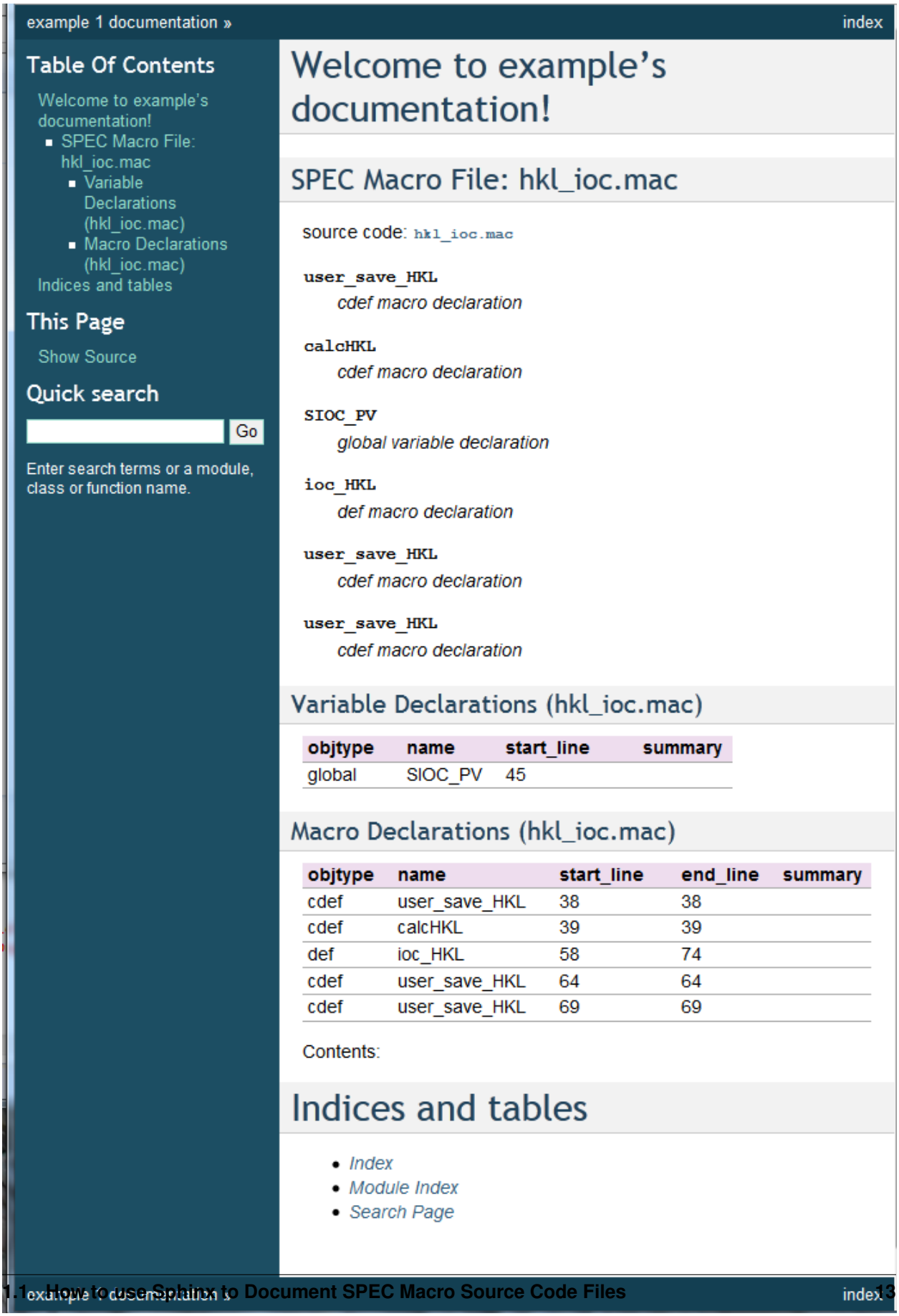

Fig. 1.5: Documentation of the original **hkl_ioc.mac** file.

```
# Summary: This macro scans the sample in a circular mesh.
#
# Dependencies: It is part of the circular mesh macro package.
```

Its literal text rendition in reST is written:

```
"""
::

        # Summary: This macro scans the sample in a circular mesh.
        #
        # Dependencies: It is part of the circular mesh macro package.

"""
```

which looks like:

```
    # Summary: This macro scans the sample in a circular mesh.
    #
    # Dependencies: It is part of the circular mesh macro package.
```

Note that the two colons indicate literal text follows. Both the blank line after the colons and the final blank line are required to avoid warnings. And, the entire comment *must be indented* at least one column to the right of the two colons.

reST markup In reST, the SPEC comment above might be written as two definition list items [2]:

```
"""
Summary
    This macro scans the sample in a circular mesh.

Dependencies
    It is part of the circular mesh macro package.

"""
```

which looks like:

Summary This macro scans the sample in a circular mesh.

Dependencies It is part of the circular mesh macro package.

(The final blank line is necessary to avoid warnings.)

This markup does not look too complicated until we reach the *Modification history* starting at line 23. The content here might be coded as either literal text (above) or a reST table. Since the table is easy *and* CVS is no longer used to build the revision history, we'll format it as a table.

Consider this SPEC comment:

[2] definition list: http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#definition-lists

```
#Revision 1.3  2006/05/22 20:34:35  jiaox
#removed unsed lines in ioc_HKL.
#
#Revision 1.2  2006/05/11 17:46:31  jiaox
#Added CVS Log entry.
```

It might be put into a table [3] such as:

```
========   ==================  =======  ====================================
Revision   date/time           author   remarks
========   ==================  =======  ====================================
1.3        2006/05/22 20:34:35  jiaox    removed unsed lines in ioc_HKL.
1.2        2006/05/11 17:46:31  jiaox    Added CVS Log entry.
========   ==================  =======  ====================================
```

which looks like:

Revision	date/time	author	remarks
1.3	2006/05/22 20:34:35	jiaox	removed unsed lines in ioc_HKL.
1.2	2006/05/11 17:46:31	jiaox	Added CVS Log entry.

Global Docstring

The convention is to treat the first docstring in a macro file as the *global docstring*.

With these ideas in mind, here is the markup of the first 32 lines:

```
"""
SPEC macros for the Advanced Photon Source

Beamline/Sector
        4ID

Macro Package
        hkl_ioc.mac

Version
        1.0 (August,2005)

Description
        A user defined calcHKL to write the current HKL postion to a
        soft IOC. It requires spec softioc running.

Written by
        X. Jiao 08/08/2005

Modified by:

User macros
        ioc_HKL -> to turn on/off the feature of putting HKL to shared
        memory.

Internal macros
```

[3] table: http://sphinx.pocoo.org/rest.html#tables

```
        ioc_put_HKL -> write HKL to the soft IOC

Modification history:

========   ==================  =======  ====================================
Revision   date/time           author   remarks
========   ==================  =======  ====================================
1.3        2006/05/22 20:34:35  jiaox    removed unsed lines in ioc_HKL.
1.2        2006/05/11 17:46:31  jiaox    Added CVS Log entry.
========   ==================  =======  ====================================
"""
```

which (except for the section title which is hard to render in this document) looks like:

Beamline/Sector 4ID

Macro Package hkl_ioc.mac

Version 1.0 (August,2005)

Description A user defined calcHKL to write the current HKL postion to a soft IOC. It requires spec softioc running.

Written by

24. Jiao 08/08/2005

Modified by:

User macros ioc_HKL -> to turn on/off the feature of putting HKL to shared memory.

Internal macros ioc_put_HKL -> write HKL to the soft IOC

Modification history:

Revision	date/time	author	remarks
1.3	2006/05/22 20:34:35	jiaox	removed unsed lines in ioc_HKL.
1.2	2006/05/11 17:46:31	jiaox	Added CVS Log entry.

Test

Be sure to test your changes as you progress, until you are confident with reST markup. The *make* process is efficient, only rebuilding the documentation from affected .rst souce file changes. Usually, this also considers changes in the .mac files. This command is usually all it takes to rebuild the HTML documentation:

```
make html
```

However, you might wish to make sure changes in the .mac files cause documentation to be rebuilt. It might be easier, although less efficient, to rebuild your HTML documentation each time using this command:

```
make clean html
```

The builds are usually very fast (seconds).

Docstring markup in each macro definition

Each of the macro definitions can be marked up to provide documentation with the definition. The convention is to supply a short one-line summary first, then additional information as appropriate.

Consider the definition for `ioc_HKL`. A summary of it is given in the first comment section. We'll apply that as the docstring:

```
def ioc_HKL '{
    """to turn on/off the feature of putting HKL to shared memory."""

    if($# != 1) { eprint "Usage: ioc_HKL on/off ";exit}


    if(("$1" == "on")) {
       cdef("user_save_HKL","ioc_put_HKL","ioc_HKL","0x20")
       print "Now put HKL to softioc."
       exit
    }
    if(("$1" == "off")) {
       cdef("user_save_HKL","","ioc_HKL","delete")
       print "Stop put HKL to softioc."
       exit
    }
    eprint "Usage: ioc_HKL on/off "
}'
```

Do the same thing for the *ioc_put_HKL* macro definition.

Document the global variable

On line 45 (in the original file), there is a global variable declaration: `global SIOC_PV`. The description for this variable has been deduced from its usage in this file with EPICS. [4] It is possible to document `SIOC_PV` using a *Descriptive comments*. Insert the line containing `global SIOC_PV` with this one:

```
global SIOC_PV              ;#: soft IOC PV name
```

The semicolon is used to ensure that the spec command is finished. It might be unnecessary. The descriptive comment can be used *in-line* to define the item on that line.

Similarly, add another *Descriptive comments* to document line 38 (original file) by inserting this line *before* the line that reads `cdef("user_save_HKL","","ioc_HKL"):`

```
#: preload check/setting here
```

Here the descriptive comment appearing on one line will provide a summary for the item defined on the next line only.

Final Results

Rebuild the completed `../markup_example/hkl_ioc.mac` documentation with:

```
make html
```

[4] EPICS: http://www.aps.anl.gov/epics

Note: Don't be concerned about the warnings of `SEVERE: Duplicate ID: "cdef-user_save_HKL".` These messages are only informative. This *bug* should be resolved in a future version of `specdomain`.

After refreshing the page in the WWW browser, it should like this:

and the index will look like:

Note: It is *planned for the future* to provide options for sorting the output alphabetically and to provide other features.

1.2 Examples

1.2.1 Simple Example to Document a SPEC Macro File

This example demonstrates how a file with simple reST markup will be documented.:

```
.. autospecmacro:: simple.mac
```

SPEC Macro File: simple.mac

source code: `simple.mac`

Example SPEC Macro Source code file for demonstration purposes only.

```
$Id$
```

This file is used to demonstrate how to document SPEC [1] macro files using Sphinx [2] and restructured text (reST) [3] markup.

Provides:

- `inc1scan`
- `CheckSaveToFile`

Footnote

`example_global`
global variable declaration

this is an example of a global variable

`A_keV`
constant declaration

Conversion constant between wavelength (A) and photon energy (keV) ($E\lambda = hc$)

[1] SPEC: http://www.certif.com
[2] Sphinx: http://sphinx.pocoo.org/
[3] reST: http://docutils.sf.net/rst.html

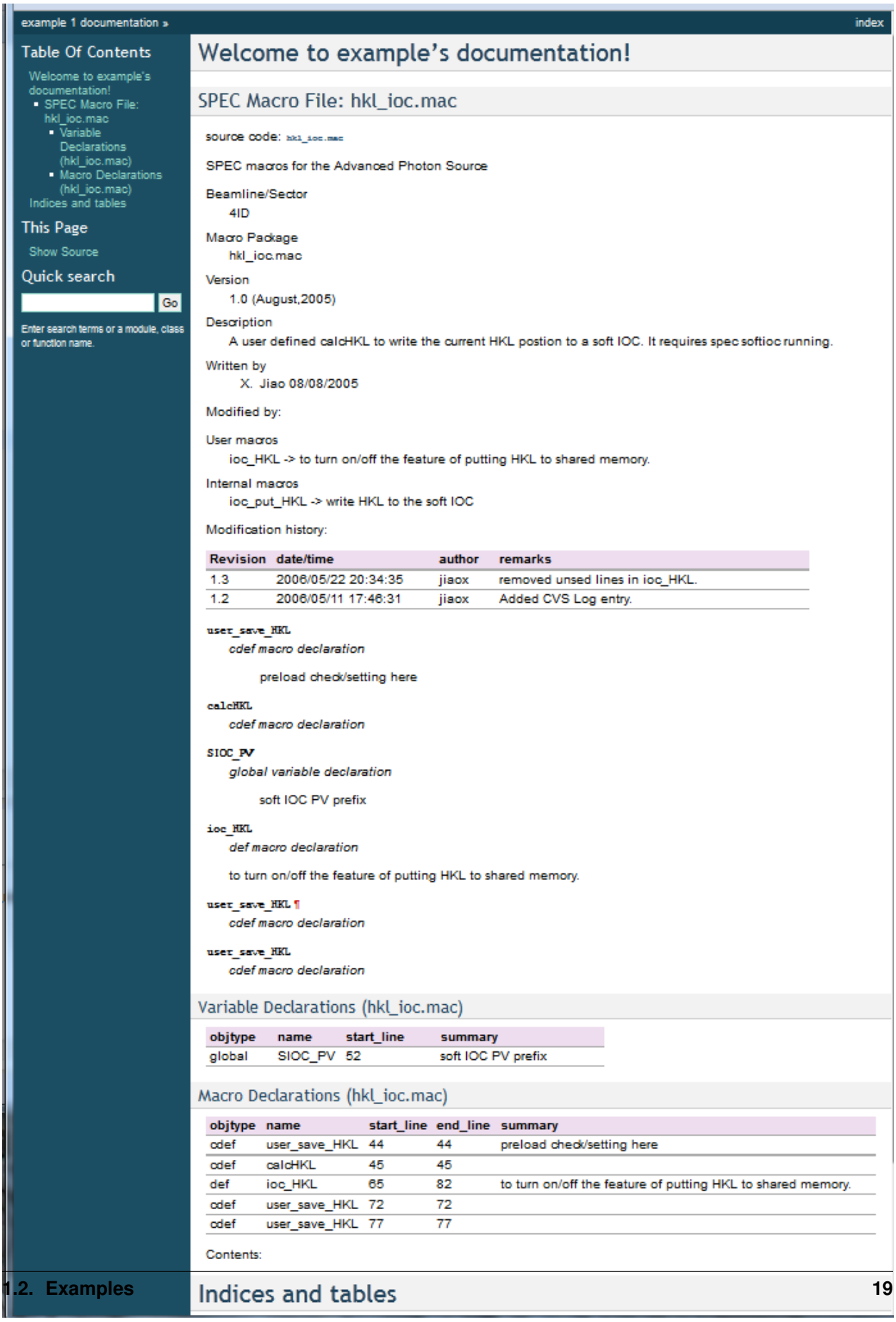

Fig. 1.6: Documentation of the marked-up **hkl_ioc.mac** file.

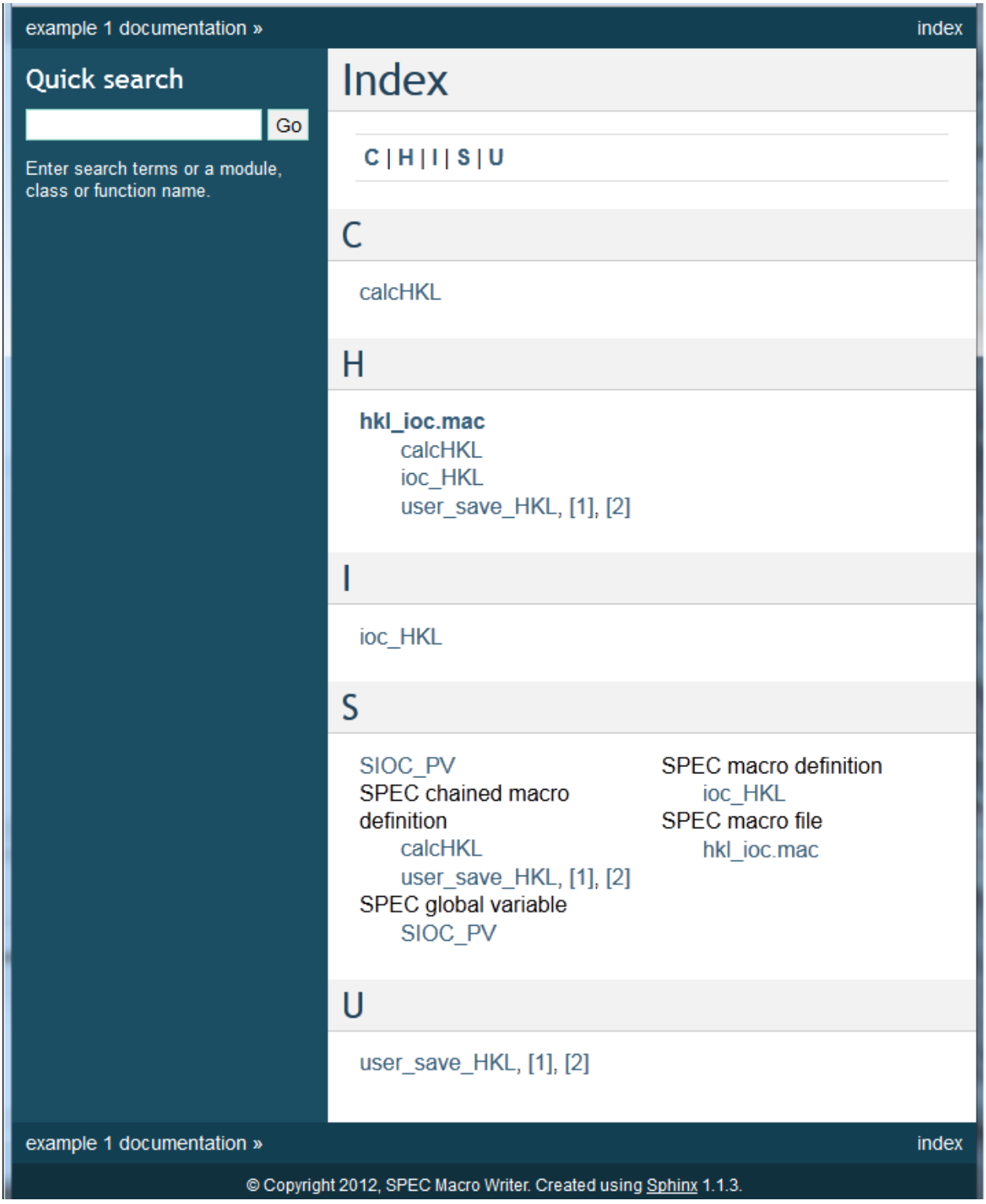

Fig. 1.7: Index of the marked-up **hkl_ioc.mac** file.

inc1scan
def macro declaration

specify a single motor scan with interval rather than # of intervals

CheckSaveToFile
def macro declaration

Helps prevent user from writing data to */dev/null* by accident.

This macro will run when SPEC starts and checks if the output is directed into a file or might be ignored (written to */dev/null*). It runs `newsample` if the output might be ignored.

begin_mac
cdef macro declaration

register CheckSaveToFile to run on startup

Variable Declarations (simple.mac)

obj-type	name	start_line	summary
global	exam-ple_global	33	this is an example of a global variable
con-stant	A_keV	37	Conversion constant between wavelength (A) and photon energy (keV) ($E\lambda = hc$)

Macro Declarations (simple.mac)

obj-type	name	start_line	end_line	summary
def	inc1scan	39	46	specify a single motor scan with interval rather than # of intervals
def	Check-SaveToFile	48	62	Helps prevent user from writing data to */dev/null* by accident.
cdef	begin_mac	64	64	register CheckSaveToFile to run on startup

1.2.2 Example with more extensive reST markup

This example demonstrates how a file more extensive reST markup will be documented.:

```
.. autospecmacro:: bpm.mac
```

SPEC Macro File: bpm.mac

source code: `bpm.mac`

Macro support for a beam position monitor.

Purpose Macro support for a beam position monitor. The BPM has a serial interface. This support uses the EPICS generic serial record as a communications channel.

Dependencies

These macros require:

EPICS Experimental Physics and Industrial Control System, http://www.aps.anl.gov/epics

EPICS support for XBPM The EPICS support for the X-ray Beam Position Monitor (XBPM) is needed to interface between the RS-232 connection on the XBPM and the EPICS IOC using an octal232 IP card. The software support consists of an EPICS generic serial record, an EPICS database, and a state notation language sequence program.

EPICS PV support prefix The support prefix is `ioc:xbpm` (configurable in `XBPM_PREFIX`)

Usage:

```
qdo /epics/clients/spec/macros/bpm.mac
```

GX_FACTOR
constant declaration

X-axis calibration, mm/unit

GY_FACTOR
constant declaration

Y-axis calibration, mm/unit

XBPM_PREFIX
constant declaration

EPICS support prefix

BPM_read_diode(u'chan',)
def() macro function declaration

read current value from the named BPM channel

param character chan channel letter (a, b, c, or d)

example:

```
value = BPM_read_diode("b")
```

BPM_set_defaults
def macro declaration

Set (or reset) proper default values on the XBPM.

Also, measure the dark signals at each amplifier gain scale (*range*) for each photodiode by calling `BPM_setAllDarkSignals`. This process takes about one minute. The software does not check to make sure that the beam is **off** so you should be sure that the shutter is closed during the time you run this macro. When finished, set the amplifier gain back to the value it had as this macro was started.

BPM_setDarkSignal(u'range',)
def() macro function declaration

measure and set the dark signal on each channel for the given amplifier range

param int range $0 <= range <= 5$

Measure and set the dark signal for each photodiode amplifier at the selected amplifier gain scale (*range*). Report the findings as a spec comment.

range	gain
0	350 nA
1	700 nA
2	1400 nA
3	7 uA
4	70 uA
5	700 uA

BPM_setAllDarkSignals
def macro declaration

measure and set the dark current for all amplifier ranges

Variable Declarations (bpm.mac)

objtype	name	start_line	summary
constant	GX_FACTOR	30	X-axis calibration, mm/unit
constant	GY_FACTOR	31	Y-axis calibration, mm/unit
constant	XBPM_PREFIX	32	EPICS support prefix

Macro Declarations (bpm.mac)

objtype	name	start_line	end_line	summary
def	BPM_set_defaults	48	92	Set (or reset) proper default values on the XBPM.
def	BPM_setAllDarkSignals	153	161	measure and set the dark current for all amplifier ranges

Function Macro Declarations (bpm.mac)

objtype	name	start_line	end_line	args	summary
function def	BPM_read_diode	34	46	chan	read current value from the named BPM channel
function def	BPM_setDarkSignal	94	151	range	measure and set the dark signal on each channel for the given amplifier range

1.2.3 Example with no reST markup

This example demonstrates how a file with no particular markup will be documented.:

```
.. autospecmacro:: aalength.mac
```

SPEC Macro File: aalength.mac

source code: `aalength.mac`

strjoin(u's,delimiter',)
def() macro function declaration

Function Macro Declarations (aalength.mac)

objtype	name	start_line	end_line	args	summary
function def	strjoin	21	27	s,delimiter	

1.3 SPEC Documentation Conventions

This document lays out several conventions for documenting SPEC macro source code files. The aim of these conventions is to to help provide consistency for the "look and feel" of the resulting documentation. However, these conventions are by no means strict requirements.

1.3.1 Documentation in comment blocks

Inline source documentation resides inside comment blocks within the SPEC macro files. It is important to note, however, that not every comment in the source code is part of the documentation. Rather, the comments containing the documentation need to be placed in a certain context, depending on the scope of the documentations. In analogy to the python language, we will refer to these documentation comments as *docstrings*, even though there are some differences concerning how they are implemented.

Comments in SPEC

There are two ways to mark a comment in SPEC. The usual identifier is the #-sign preceding a comment:

```
# This is a single comment line

my_val = 2.0  # This is an in-line comment
```

A less well-known identifier to designate multi-line comments is the use of triple double-quotes (`"""`), which were introduced specifically with docstrings in mind [1]:

```
"""
This is an extended comment in SPEC.

Note that it can span multiple lines and contain several paragraphs.

"""
```

Warning:
Do not use the single-quote characters (`'`) to mark an extended comment! In the Python language, a docstring can be included either in triple double-quotes (`"""`) or in triple single-quotes (`'''`). But, **unlike Python**, SPEC does not recognize single quotes to mark extended comments. Only use the double quote character for SPEC files.

Extended comments

The first extended comment in a "section" should contain the docstring. Any other extended comments will be ignored and not processed during the documentation creation (this setting could be changed with an optional switch.) In this context, a *section* refers to a particular "code object", which might be the global scope of a .mac file or a macro definition block, for example.

[1] SPEC extended comments for docstrings: http://www.certif.com/spec_help/chg5_01.html

The first paragraph of the docstring should be a concise summary line, followed by a blank line. This summary will be parsed in a special way to be included as a description of the code object in summary tables, indices, etc. If the first paragraph starts with a colon (`:`), no summary text will be assumed.

Following the summary, a more elaborate description of the code object may be given.

For macro definitions (`def, rdef`), the docstring should immediately follow the declaration line and be indented to the same level as the code contained within the definition. It is also recommended to insert a blank line between the last paragraph in a multi-line docstring and its closing quotes, placing the closing quotes on a line by themselves:

```
def my_macro_def '{
  """
  This is the summary line.

  And here is some more elaborate discussion of the functionality, which may
  again extend over several lines or paragraphs, and contain all the required
  reST and Sphinx markup.

  """

  my_var = 1.0

  # do some more stuff...

}'
```

Finally, it is recommended to use the extended comment syntax with triple-quotes only for docstrings, even though it is a valid syntax to include longer blocks of comments about the code itself. To avoid confusion between the two types of comments, non-documentation comments should be included by preceding each line with the `#`-sign:

```
"""
This is my docstring.

"""

# Here, I write down some
# comments about how
# exactly my code works:
#
# Increment x by 1 for each registered photon

if(hit) x+=1
```

Descriptive comments

Caution: This is new convention, yet it does not violate any SPEC rules. It *is* awfully useful!

Descriptive comments are a new construct which can be used to document items that cannot contain extended comments (triple-quoted strings) themselves, such as variable declarations or *rdef* or *cdef* macro declarations. (They can also be used to document one-line *def* macros!) They appear either as comments in the same line after the declaration (in-line) or as a comment-only line immediately preceding the declaration (one-liner). Descriptive comments are marked by a preceding `#:`, which lets them appear like normal SPEC comments, but the colon triggers the parser to process the docstring.

Like the summary lines in extended comments, these descriptive comments are used as descriptions in summary tables, etc.

Examples:

Descriptive comment that documents **TTH**, a global variable declaration:

```
global TTH    #: two-theta, the scattering angle
```

Descriptive comment that documents **ccdset_shutter**, an *rdef* declaration:

```
#: clear the ccd shutter handler
rdef ccdset_shutter ''
```

Descriptive comment that documents **do_nothing()**, a *function def* declaration:

```
def do_nothing() ''      #: this macro does do anything
```

Hidden objects

Hidden objects begin with at least one underline character, such as `_hidden`. This includes macros and variables. These should be optional in the documentation.

Anonymous objects begin with at least two underline characters, such as `___anon`. This includes macros and variables. These should not be documented unless specifically requested and only then if hidden objects are documented.

Undeclared variables

Undeclared variables (those with no formal global, local, constant, or array declaration) will not be documented. At least for now.

Parameter descriptions

Use the same syntax as parameter declarations for Python modules. Here is an example SPEC macro with reST markup:

```
def my_comment '{
        """
        Make a comment

        USAGE::

          > my_comment "AR aligned to 15.14063 degrees"

        ARGUMENTS:

          :param str text: message to be printed

        """

        qcomment "%s" $1
}'
```

which documentation looks like this:

my_comment

Make a comment

USAGE:

```
        > my_comment "AR aligned to 15.14063 degrees"``
```

ARGUMENTS:

param str text message to be printed

arg float x some number to be processed

arg y another number (without type)

RETURNS:

Returns str comment the comment

demo_cdef_more['<<< cdef argument list not handled yet >>>']

This is my punch line!

param str demo_cdef_more name of chained macro

param str spec_code SPEC code to be executed (usually a single macro name)

param str key name of this part of the chained macro

param flags see **SPEC** documentation for details

param str not_here something is missing...

rtype none

1.4 SPEC Documentation Style Guide

Some interesting and applicable words from the Google Style Guide:

> BE CONSISTENT.
>
> If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.
>
> The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

With these words in mind, this SPEC Documentation Style Guide documents the conventions set forth to use for SPEC macros at the APS.

1.4.1 The concept of docstrings

In-line source documentation resides inside comment blocks directly within the SPEC macro files. In analogy to the python language, we will refer to these documentation comments as *docstrings*. These docstrings are processed by the specdomain package for the Sphinx documentation creator to produce user or reference manual in a variety of formats (html, pdf, man-pages, text files, etc.)

The following section sets forth some formatting conventions for SPEC docstrings.

One-line docstrings

There are two distinct scenarios where one-line docstrings are appropriate or even necessary:

1. Obvious cases (where one line is completely sufficient to describe the code).
2. Descriptive comments, that are used to document code objects (variables, one-line `rdef` or `def` declarations, or `'cdef` definitions) which cannot contain extended docstrings.

Obvious cases Obvious cases are those where the macro or macro function definition is so clear that no elaborate explanation is necessary. For example, the following macro function definition can probably be documented in a one-liner:

```
def sind(x) '{
  """ Return the sine of x, where x is given in degrees."""

  return sin(x*PI/180)
}'
```

One-liners need to be enclosed in triple double-quotes (`"""`) which are placed on the same line as the docstring. A single space between the opening quotes and the docstring is optional. A blank line after the docstring helps to visually separate it from the actual code.

Descriptive comments Descriptive comments are a new a construct which can be used to document items that cannot contain extended comments (triple-quoted strings) themselves, such as variable declarations, one-line `def` or `rdef` declarations, or `cdef` definitions. They appear either as comments in the same line after the declaration (in-line) or as a comment-only line immediately preceding the declaration (one-liner). Descriptive comments are marked by a preceding `#:`, which lets them appear like normal SPEC comments, but the colon triggers the parser to process the docstring:

```
global TTH            #: The scattering angle two-theta [float].

#: Clear the ccd shutter handler
rdef ccdset_shutter ''

def do_nothing() ''   #: This macro does not do anything.
```

Multi-line docstrings

Multi-line docstrings are surrounded by a pair of triple double-quotes (`"""`), which should be placed on a line by themselves. For macro definitions, the opening quotes should appear on the next line immediately below the macro definition. It is recommended to insert a blank line between the last paragraph in a multi-line docstring and its closing quotes, followed by another blank line before the next code item begins.

The entire docstring is indented the same as the quotes at its first line. Docstrings inside macro declarations should be indented from the definition statement by the same level as the code contained in the definition.

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line and then a more elaborate description. The summary line will be used by the specdomain indexing and summary tools. It is therefore important to make the summary lines very clear and concise. They should always be written as complete sentences, starting with a capital letter and ending with a period.

1.4.2 Documentation of code objects

We will refer to certain types or components of the SPEC macro code as *code objects*. These may include:

- Macro files

- Macro definitions (`def`, `rdef`, `cdef`)
- Variables (global, local, etc.)
- Entire collections of macro files

Each type of these code objects requires certain information to be included in the documentation. The following sections should help to ensure that all the required information is included and will appear in a consistent format.

File headers

The macro file header docstring provides information about the macro file as a whole (in the python world, this might be called a *module*).

As with any docstring, the first item should be a concise summary line of what the macro file provides and which could be used in summary tables, indexes, etc.

This is followed by sections about the detailed functionality, setup and configuration instructions, file information, and so on. The full power of Sphinx and ReST markup is available at this level, so sections can be broken up in subsections and subsubsections, tables may be included as well as figures or mathematical formulas.

The following information should be included, and the below layout may aid in supplying a complete set of information. Note that this can always be changed to meet the particular requirements of individual macro files:

Description (top-level header): A more elaborate description of the functionality provided in the macro file. Include any number of subsections and subsubsections.

Notes (top-level header): Any additional notes or comments about the file or its usage.

Installation (top-level header): Information on how to set up the macro functionality. This includes, if applicable, the following subsections (second level headers):

Configuration: Prerequisites in the SPEC configuration. For example, the configuration of dedicated counters may be necessary in order to use the macros.

Setup: The steps necessary to set up the macro functionality. For example, loading the macro file (`qdo`) and running the `macro_init` function.

Dependencies: List all the dependencies on other macros, hardware, software, EPICS channels, etc.

Impact: Describe the impact that the use of the macro may have. For example, list all the changes made to other `cdef` macro definitions by this macro file.

File Information (top-level header): All the information about the macro file itself, like authors, license, version, etc.

It is recommended to build up this section as a definition list. The headings for each item are CAPITALIZED and end with a colon. The content under each of these items should be indented one level. This results in a more lightweight layout, and prevents cluttering the tables of content with too many subsections.

The following items should be included, preferably in this order:

- AUTHOR(S):
- CREATION DATE:
- COPYRIGHT:
- LICENSE:
- VERSION:
- CHANGE LOG:
- TO DO:

- KNOWN BUGS:

See the example below for more details on each of these items.

Example of a file header docstring

```
"""
Summary line: a concise sentence about what this macro file provides.

Description
===========
A more detailed description of the macro file and the functionality that the
library of macro definitions it contains provides.

Note(s)
=======
Any special notes about the macro file, its usage, or its history can go here.

Installation
============
Describe, as applicable, the installation procedure, necessary changes in the
SPEC configuration, dependencies, and impact on chained macro definitions
(``cdef``) or redefinitions (``rdef``). The sections below give hypothetical
examples of what the content may look like.

Configuration
-------------
For this macro to work, the SPEC configuration may need to be modified. The
following counters are required:

        ========  ============  ==============  ======================
        Counter   Mnemonic      Type            Description
        ========  ============  ==============  ======================
        mycount1  mcnt1         None            My first counter
        mycount2  mcnt2         Macro counter   My second counter
        ========  ============  ==============  ======================

Setup
-----
With the above configuration change, simply load the macro file
``template.mac`` and run :spec:def:`template_setup`::

        > qdo template.mac
        > template_setup

Dependencies
------------
This macro depends on the following macro files and macros:

* filter.mac

        - :spec:def:`filter_trans`
        - :spec:def:`filter_get_trans()`

* bpm.mac

        - :spec:def:`bpm_get_pos`
```

```
Impact
------
The following chained macro definitions are affected by template.mac:

* :spec:def:`user_precount`:  Adding :spec:def:`template_precount`
* :spec:def:`user_getcounts`: Adding :spec:def:`template_getcounts`
        to the end ``(0x20)``

File information
================

AUTHOR(S):

        * A.B. Sample (AS, asamp), asamp@aps.anl.gov, Argonne National Laboratory

CREATION DATE:

        YYYY-MM-DD

COPYRIGHT:

        .. automatically retrieve the current year:
        .. |current_year| date:: %Y

        Copyright (c) 2010-|current_year|, UChicago Argonne, LLC
        Operator of Argonne National Laboratory
        All Rights Reserved

        APS SPEC macros

LICENSE::

        OPEN SOURCE LICENSE

        Redistribution and use in source and binary forms, with or without
        modification, are permitted provided that the following conditions are met:

        1. Redistributions of source code must retain the above copyright notice,
                this list of conditions and the following disclaimer.  Software changes,
                modifications, or derivative works, should be noted with comments and
                the author and organization's name.

        2. Redistributions in binary form must reproduce the above copyright notice,
                this list of conditions and the following disclaimer in the documentation
                and/or other materials provided with the distribution.

        3. Neither the names of UChicago Argonne, LLC or the Department of Energy
                nor the names of its contributors may be used to endorse or promote
                products derived from this software without specific prior written
                permission.

        4. The software and the end-user documentation included with the
                redistribution, if any, must include the following acknowledgment:

                "This product includes software produced by UChicago Argonne, LLC
                under Contract No. DE-AC02-06CH11357 with the Department of Energy."

        ****************************************************************************
```

```
        DISCLAIMER

        THE SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND.

        Neither the United States GOVERNMENT, nor the United States Department
        of Energy, NOR UChicago Argonne, LLC, nor any of their employees, makes
        any warranty, express or implied, or assumes any legal liability or
        responsibility for the accuracy, completeness, or usefulness of any
        information, data, apparatus, product, or process disclosed, or
        represents that its use would not infringe privately owned rights.

        ********************************************************************************

VERSION::

        $Revision$
        $Date$
        $Author$
        $URL$

CHANGE LOG:

        YYYY-MM-DD (AS):

        - created first version of this macro.
        - tested on a dummy SPEC version not connected to a diffractometer.

        YYYY-MM-DD (AS):

        - added a new macro definition: :spec:def:`new_macro` to display the status.

TO DO:

        - List all the TODO items

KNOWN BUGS:

        - List all the known bugs and limitations

"""
```

Macro definition docstrings

The docstring for a macro or macro function definition should summarize its behavior and document its arguments, return value(s), side effects, and restrictions on when it can be called (all if applicable). A docstring should give enough information to write a call to the function without reading the function's code. A docstring should describe the function's calling syntax and its semantics, not its implementation.

Certain aspects of a macro definition should be documented in special sections, listed below. Since Sphinx does not generally allow for the presence of any types of formal headings inside the code object docstrings, the docstring should be build up as a ReST definition list (see example below). The section titles are all CAPITALIZED for improved visibility and end with a colon. The contents for each section are indented by two spaces with respect to the section title.

Sections

The following sections should be included in the macro docstring after the summary line, in the below order, if applicable:

DESCRIPTION: A more elaborate description of the macro's functionality.

USAGE: The syntax for calling the macro. This should contain all possible variants of the macro call. Argument names are enclosed in angle brackets (<>) to indicate that they should be replaced by actual values in the macro call. Optional arguments are additionally enclosed in square brackets ([]). The actual USAGE syntax should appear as pre-formatted text, and each input line should start with a ">"-symbol to represent the SPEC command line prompt:

```
USAGE::

  > my_macro <pos1> [<pos2>]
  > <return_value> = my_function(<input_value>)
```

ARGUMENTS: All the arguments to a function or macro call should be listed in the form of a ReST field list. The argument name is enclosed between colons (:), followed by the description, which can span several (indented) lines. It is useful to specify also the type of the argument in square brackets ([]).

If a macro call has both mandatory and optional arguments, list them in separate lists:

```
Required arguments:
  :pos1:    Target position for motor [float].

Optional arguments:
  :timeout: The wait-time before giving up on serial port communication in
    seconds [float].
```

Note that a number of python projects use a special kind of argument definition list which is processed by Sphinx to include more information, for example, the type of an argument. Other projects, however, actively discourage its use or prefer the above style for simplicity. The syntax is as follows:

```
:param str motor_name: name of motor to use.
```

This syntax is perfectly acceptable also for SPEC documentation, however it arguably results in harder to read in-line documentation and is often not rendered very neatly in the final Sphinx output. Use this at your own discretion.

EXAMPLE: A short example, illustrating the usage of the macro. As in the case of the USAGE section, the syntax should appear as pre-formatted text, and each input line should start with the ">"-symbol to represent the SPEC command line prompt. Short explanation lines can be inserted as indented comment lines:

```
EXAMPLE::

  > set_temperatures 23.5 50.0
      # sets the two container temperatures to 23.5 and 50.0 degrees.
```

NOTE(S): Additional notes on the macro usage.

SEE ALSO: A list of other macros or documentation items to refer to for further information. If possible, these should be dynamically linked using the corresponding Sphinx specdomain roles:

```
SEE ALSO:
  * :spec:def:`my_other_macro`
  * http://spec.examples.com/example3.html
```

Using the definition list syntax, other sections may be included, as necessary or appropriate for the particular macro.

Example of a macro definition docstring

```
"""
Concise summary line.

USAGE::

        > my_move <motor> <position> [<sleep_time>]

ARGUMENTS:

        Required arguments:
                :motor:    The motor to be moved [str].
                :position: The position to move the motor to [float].

        Optional arguments:
                :sleep_time: Settling time after the move has finished [float].

EXAMPLE::

        > my_move del 23.2346 0.3
                        # move del to 23.2346 and wait for 0.3 seconds after move finishes.

NOTE:
        Indicate any side effects, restrictions or other usage notes here.

SEE ALSO:
        * :spec:def:`my_move2`
        * :spec:global:`MOVE_FLAG`
        * http://www.certif.com/spec_help/prdef.html

"""
```

This results in the following:

Concise summary line.

USAGE:

```
> my_move <motor> <position> [<sleep_time>]
```

ARGUMENTS:

Required arguments:

motor The motor to be moved [str].

position The position to move the motor to [float].

Optional arguments:

sleep_time Settling time after the move has finished [float].

EXAMPLE:

```
> my_move del 23.2346 0.3
                # move del to 23.2346 and wait for 0.3 seconds after move finishes.
```

NOTE: Indicate any side effects, restrictions or other usage notes here.

SEE ALSO:

- my_move2

- `MOVE_FLAG`
- http://www.certif.com/spec_help/prdef.html

One-line docstrings

As mentioned previously, one-line docstrings (also called *descriptive comments*) can be used to document code objects that cannot contain extended docstrings.

One-line docstrings begin with a capital letter and end with a period.

Variables

Docstrings for variables provide a short description of the variable. It is also recommended to specify the type of the variable in square brackets (`[]`). For example:

```
global TTH     #: The scattering angle two-theta [float].
local _ind     #: List index of the active reflection [int].

#: Associate array with orientation reflection HKL-indices & angles [float].
global ORIENTATION_REFLECTIONS
```

Macro definitions

One-line docstrings for macro definitions contain a short description of the purpose for the (re-)definition. For example:

```
#: Define the ccd shutter handler
rdef ccdset_shutter '_ccdset_shutter'

#: remove ccd_getcounts from user_getcounts
cdef("user_getcounts", "", "ccd_key", "delete")
```

Macro collection docstrings

As of now, there is no standard yet for documenting entire collections of macros, as, for example, those collected in particular directories of the SVN source code repository.

The documentation for such a collection should be in the form of normal ReST files (`*.rst`), residing in the same directory with the macro collection. There is no way of automatically including this information in the global documents yet, so it will need to be added manually somewhere in the documentation tree (at least in the global `index.rst` file or some other file that is included from the global scope).

1.5 Documenting SPEC Code

1.5.1 Documenting SPEC File Directories

`.. autospecdir:: path`

Param str path path (absolute or relative to the .rst file) to an accessible directory with SPEC macro files

```
.. autospecdir:: /home/user/spec.d/macros
```

Note: options planned for future versions:

- document all subdirectories of *path*
- match files by a pattern (? glob *v.* re ?)

1.5.2 Documenting SPEC Files

`.. autospecmacro:: filename`

Param str filename name (with optional path) of the SPEC macro file. The path component can be relative or absolute. If relative, the path is relative to the *.rst* file.

Document the contents of a SPEC macro source code file, including extended comments, *def*, *rdef*, and *cdef* macro declarations and changes, and *local*, *global*, and *constant* variable declarations.

```
.. autospecmacro:: sixc.mac
```

Note: options planned for future versions:

- standard documentation pattern
- include all extended comments
- ignore hidden objects [1]
- ignore anonymous objects [2]

1.5.3 SPEC Extended Comments

Do not use the single-quote character to mark an extended comment.

In the Python language, a triple-quoted string is known as a docstring. But, **unlike Python**, SPEC does not recognize single quotes to mark extended comments. Only use the double quote character for SPEC files.

Since 2002, SPEC has provided an *extended comment* block. [3] Such a block begins and ends with three double-quotes, such as:

```
"""This is an extended comment"""
```

Here, the extended comment block should be formatted according to the conventions of restructured text [4]. There is also a *recommended convention* for using extended comments in SPEC macro files.

[1] *hidden* objects begin with one underline character, such as `_hidden`
[2] *anonymous* objects begin with at least two underline characters, such as `__anon`
[3] SPEC extended comments: http://www.certif.com/spec_help/chg5_01.html
[4] restructured text: http://docutils.sf.net/rst.html

1.5.4 Describing SPEC objects

The following *directives* refer to objects in SPEC macro source code files and create index entries and identifiers:

`.. spec:def::`
Declare the name of a SPEC `def` macro.

ReST code	Displays like
`.. spec:def:: demo_def`	**demo_def**

`.. spec:rdef::`
Declare the name of a SPEC `rdef` run-time-defined macro.

ReST code	Displays like
`.. spec:rdef:: demo_rdef`	**demo_rdef**

`.. spec:cdef::`
Declare the name of a SPEC `cdef` chained macro.

ReST code	Displays like
`.. spec:cdef:: cdef("demo_cdef")`	**demo_cdef**['<<< cdef argument list not handled ye

More elaborate example showing how to call a chained macro and also describe the arguments:

```
.. spec:cdef:: cdef("demo_cdef_more", "spec_code", "key", flags)

   :param str demo_cdef_more: name of chained macro
   :param str spec_code: SPEC code to be executed (usually a single macro name)
   :param str key: name of this part of the chained macro
   :param flags: see http://www.certif.com/spec_help/funcs.html
   :rtype: none

   This text is ignored (for now).
```

Displays like:

demo_cdef_more['<<< cdef argument list not handled yet >>>']

Parameters

- **demo_cdef_more** (*str*) – name of chained macro
- **spec_code** (*str*) – SPEC code to be executed (usually a single macro name)
- **key** (*str*) – name of this part of the chained macro
- **flags** – see **SPEC** documentation for details

Return type none

`.. spec:global::`
Declare the name of a SPEC global variable.

ReST code	Displays like
`.. spec:global:: demo_global`	**demo_global**

.. spec:local::
Declare the name of a SPEC local variable.

ReST code	Displays like
`.. spec:local:: demo_local`	**demo_local**

.. spec:constant::
Declare the name of a SPEC constant.

ReST code	Displays like
`.. spec:constant:: demo_const`	**demo_const**

1.5.5 Cross-referencing SPEC objects

The following *roles* refer to objects in SPEC macro source code files and are possibly hyperlinked if a matching identifier is found:

:spec:def:
Reference a SPEC macro definition by name. (Do not include the argument list.)

```
An example ``def`` macro: :spec:def:`demo_def`
```

An example `def` macro: `demo_def`

:spec:rdef:
Reference a SPEC run-time macro definition by name. (Do not include the argument list.)

```
An example ``rdef`` macro: :spec:rdef:`demo_rdef`
```

An example `rdef` macro: `demo_rdef`

:spec:cdef:
Reference a SPEC chained macro definition by name. (Do not include the argument list.)

```
An example ``cdef`` macro: :spec:cdef:`cdef("demo_cdef")`
An example ``cdef`` macro: :spec:cdef:`cdef("demo_cdef_more")`.
```

An example `cdef` macro: `cdef("demo_cdef")`. An example `cdef` macro: `cdef("demo_cdef_more")`.

:spec:global:
Reference a global-scope variable.

```
An example ``global`` variable: :spec:global:`demo_global`
```

An example `global` variable: `demo_global`

:spec:local:
Reference a local-scope variable.

```
An example ``local`` variable: :spec:local:`demo_local`
```

An example `local` variable: `demo_local`

:spec:constant:
Reference a local-scope variable.

```
An example ``local`` variable: :spec:constant:`demo_constant`
```

An example `local` variable: `demo_constant`

1.5.6 Undeclared Variables

Undeclared variables (those with no formal global, local, constant, or array declaration) will not be documented. At least for now.

1.6 Other matters

1.6.1 Downloads

home http://specdomain.readthedocs.org

svn trunk https://subversion.xray.aps.anl.gov/bcdaext/specdomain/trunk/src/specdomain/

svn source releases

- https://subversion.xray.aps.anl.gov/bcdaext/specdomain/tags/src/

PyPI http://pypi.python.org/sphinxcontrib-specdomain

1.6.2 Objectives

1. Document SPEC macro source code files
1. macro definitions (def, rdef, and cdef)
2. function definitions
3. global variable definitions
4. local variable definitions
5. array variable definitions
6. constant definitions
7. ReST-formatted comments (usually contain API or usage info)
8. optional source code highlighting
2. Build cross-referenced WWW site and PDF documentation
3. Provide a test file for development.
4. Provide useful examples in documentation.

1.6.3 TO-DO List

- move this list to the **TRAC** site: https://subversion.xray.aps.anl.gov/trac/bcdaext/wiki/SpecDomain
- add support to document SPEC array declarations
- improve the display of directives for variables and macros

- make the Index look better
- provide links from each macro to its source code
- provide for the directive options indicated in the source code
- resolve the items marked *FIXME:* in the source code
- resolve the items marked *TODO:* in the source code
- identify and document undeclared variables in the global scope
- support the :summary: role to explicitly declare a summary
- cdef argument list not handled yet
- Option to index all symbols in a macro file
- allow section headings inside macro declaration docstrings
- create a role (or example using a ref) to refer to a macro file from the documentation
- produce a custom module index with links and summary lines

1.6.4 Known bugs

- move this list to the **TRAC** site: https://subversion.xray.aps.anl.gov/trac/bcdaext/wiki/SpecDomain
- Duplicate ID warnings, for now, ignore them, the warning will be resolved in a future revision
- roles should link to directives, see *example.mac* to illustrate the problem
- fix the signature recognition for roles
- fix the signature handling for roles and directives
- the out-of-source example PNG shows a Python project, should be a SPEC project

1.6.5 Changelog

This file describes user-visible changes between the extension versions. Refer to the TRAC site for details on the tickets noting *specdomain* in the title. (https://subversion.xray.aps.anl.gov/trac/bcdaext/report/1?sort=ticket&asc=1)

Version 1.04.01 (2014-03-10)

- refs #63, #66: fixes download role of macro source code, now available on PyPI as **sphinxcontrib-specdomain**

Version 1.03 (2012-10-01)

- fixes #33: style guide has been changed to use the YYYY-MM-DD date format
- fixes #35: corrected detection of single-line rdef functions
- fixes #36: updated LICENSE text
- fixes #37: function rdef with empty arg list classified incorrectly

Version 1.02 (2012-09-09)

- contributions to documentation by Christian Schlepuetz

Version 1.01 (2012-08-15)

- refs #13: autospecdir: show a bullet list for each file at the top of the page (alternative to separate pages for each macro file)
- fixed #14: "duplicate ID warnings" resolved
- fixes #31: documentation copyright should match LICENSE file
- fixed #32: descriptive comments also for *def* definitions

Version 1.0 (2012-07-16)

- Initial release.

1.6.6 License

```
Copyright (c) 2011-2014, UChicago Argonne, LLC
Operator of Argonne National Laboratory
All Rights Reserved

specdomain


OPEN SOURCE LICENSE

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.  Software changes,
   modifications, or derivative works, should be noted with comments and
   the author and organization's name.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the names of UChicago Argonne, LLC or the Department of Energy
   nor the names of its contributors may be used to endorse or promote
   products derived from this software without specific prior written
   permission.

4. The software and the end-user documentation included with the
   redistribution, if any, must include the following acknowledgment:

   "This product includes software produced by UChicago Argonne, LLC
   under Contract No. DE-AC02-06CH11357 with the Department of Energy."

*****************************************************************************

DISCLAIMER

THE SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND.

Neither the United States GOVERNMENT, nor the United States Department
of Energy, NOR UChicago Argonne, LLC, nor any of their employees, makes
```

```
any warranty, express or implied, or assumes any legal liability or
responsibility for the accuracy, completeness, or usefulness of any
information, data, apparatus, product, or process disclosed, or
represents that its use would not infringe privately owned rights.

*******************************************************************************
```

1.6.7 Authors

Pete Jemian <jemian@anl.gov>, BCDA, Advanced Photon Source, Argonne National Laboratory

Cross-reference:

- genindex
- search

1.7 References

1.7.1 Sphinx documentation

The Sphinx website:

- **The Sphinx home page** http://sphinx.pocoo.org/index.html
- **The Sphinx documentation page** http://sphinx.pocoo.org/contents.html
- **Sphinx markup constructs** http://sphinx.pocoo.org/markup/index.html

Other helpful links:

- **Sphinx example and quickstart of pypi:** http://packages.python.org/an_example_pypi_project/sphinx.html
- **Another overview of Sphinx commands:** http://docs.geoserver.org/trunk/en/docguide/sphinx.html

1.7.2 reStructuredText documentation

- **reStructuredText home page** http://docutils.sourceforge.net/rst.html
- **reStructuredText quick reference** http://docutils.sourceforge.net/docs/user/rst/quickref.html
- **reStructuredText cheat sheet** http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt
- **reStructuredText directives** http://docutils.sourceforge.net/docs/ref/rst/directives.html

X