Reference Manual

For Spexygen 2.2.2

Document: DOC_MAN_SPX

https://www.state-machine.com
info@state-machine.com

Chapter 1

Spexygen

Remarks

- This *Spexygen* documentation has been created with *Spexygen*.
- *Spexygen* is available on `GitHub under the permissive MIT open-source license`↑. spexygen

1.1 About Spexygen

`Spexygen`↑ is a `Doxygen`↑ extension for creating traceable technical specifications, such as:

- traceable requirement specifications
- traceable source code
- traceable tests
- traceable specifications of other kind

Spexygen video↑

Note

By extending Doxygen with a uniform **traceability management** not just for source code, but also for all other specifications, *Spexygen* supports regulatory compliance with functional safety standards such as IEC 61508, IEC 62304, ISO 26262 and others.

1.1.1 Spexygen Features

The main objectives and features of *Spexygen* are:

- uniform management of traceability within all documents in the system, including source code
- provision of commands for creating well-structured, uniformely formatted, fully traceable "work artifacts"
- automating the generation of forward traceability links in the documentation (*Spexygen* generates recursive forward traceability enabling impact analysis to identify the potential consequences of a change of a given artifact)
- automating the generation of brief descriptions for the backward traceability links
- enabling DRY documentation (designed according to the "Don't Repeat Yourself" principle) by eliminating repetitions in specifying dependencies among "work artifacts"
- genearting cross-linked, **searchable**, nicely formatted documentation in HTML (modern Doxygen-awesome HTML style↑)
- genearting cross-linked, nicely formatted documentation in PDF (modern LaTeX template)
- representing documentation in human-readable text files, which can be stored in any version control system (VCS).

1.2 Traceability

Traceability is the cornerstone of any formal documentation system, especially those intended for managing **functional safety**. It enables product teams to associate every work artifact (e.g., a specific requirement) with all the related project artifacts (e.g., design, code, or tests), both backward and forward. Traceability lets everyone to see how every work artifact relates to the requirement — and vice versa — at any point during development. This ability fosters team collaboration and enables early detection of possible production risks.

Attention

Spexygen provides consistent and **automated** management of traceability within the whole documentation set.

1.2.1 Unique Identifiers (UIDs)

Traceability is enabled by the consistent use of **Unique Identifiers (UIDs)**, which are short text labels associated with *all* work artifacts, such as requirements, architecture elements, design elements, coding standard deviations, tests, functional safety documents, etc.

Note

The structure of UIDs is **flexible** to accommodate various existing naming conventions. But for compatibility with the widest possible range of cross-referencing systems and tools, the UIDs are **restricted** to generally follow the *rules for identifiers in programming languages*, such as identifiers in C, C++, or Python. Specifically, valid UIDs can contain only upper-case letters (A..Z), numbers (0..9), and underscores ('_'). Among others, UIDs cannot contain spaces, punctuation, parentheses, or any special characters like !, @, #, $, etc.

Remarks

Restricting the UIDs to the programming language identifiers allows you to *use the UIDs as identifiers*. For example, you might name test functions as their UIDs. Additionally, such UIDs become **searchable** with the Doxygen built-in search (the "Search Box").

The most important feature of UIDs is their **uniqueness** within the whole system under consideration. To avoid name conflicts, it is advantageous to establish general rules for constructing the UIDs. Throughout *Spexygen* documentation, the UIDs have the general structure consisting of fields separated by underscores:

```
 +------------------ [1] Work artifact class (e.g., 'SRS' for Software Requirement Specification)
 |   +-------------- [2] Component unique identifier (e.g, 'PRJ' for my "Project")
 |   |   +---------- [3] Work artifact ID (abbreviation or number)
 |   |   |  +------- [4] Work artifact number
 |   |   |  |   +--- [5] Optional variant letter ('A', 'B', 'C'...)
 |   |   |  |   |+-- [6] Optional version number (1, 2, 3...)
 |   |   |  |   ||
SRS_xxx_yyy_zz[_A2]
```

Examples: SRS_PRJ_Foo_01 , TUN_PRJ_free_fun_00

The various fields in the UID are as follows:

`[1]` the UID starts with a fields corresponding to the *class* of the work artifact. Here are the suggested artifact class names:

- **DOC** Document
- **SRS** Software Requirement Specification
- **SSR** Software Safety Requirement
- **SAS** Software Architecture Specification
- **SDS** Software Design Specification
- **FSM** Functional Safety Management artifact
- **SHR** Software Hazard and Risk artifact
- **DVR** Deviation Record (e.g. coding standard violation)
- **DVP** Deviation Permit (e.g. coding standard violation)
- **TUN** Test (unit)
- **TIN** Test (integration)
- **TAC** Test (acceptance)

`[2]` the Component Unique Identifier (CUI), which should be unique enough to avoid name conflicts with other software components in a larger system.

`[3]` "Work artifact ID" field identifies the artifact within the "work artifact class" `[1]`. This is the most flexible part in the UID to accommodate other existing conventions, such as MISRA deviations, the `work artifact ID` field should be easily identifiable with the MISRA Rule/Directive ID, such as `D04_01` for "Directive 4.1", or `R10_04` for "Rule 10.4". Still, please note that the more structured UID convention of using two-digits for feature groups (e.g., `D04_10` instead of `D4_10`) provide additional benefits, such as correct order under a simple alphabetical sort. This property is missing in the original MISRA IDs (e.g., a simple alphabetical sort will place Rule 10.8 *before* Rule 8.10).

`[4]` "Work artifact number" filed identifies the aspect of the work artifact

`[5]` optionally, the UID might contain a variant letter ('A', 'B', 'C',...)

`[6]` optionally, the UID might end with a single-digit version number (0..9).

Alternatively, UIDs of **code** elements follow the rules established by Doxygen, with the following general form:

```
     +--------------- [1] Namespace (e.g., class or module)
     |          +--- [2] element (e.g., attribute or operation)
     |          |
[namespace]_[element]
```

Examples: Foo_ctor() , TUN_PRJ_Foo_ctor_01

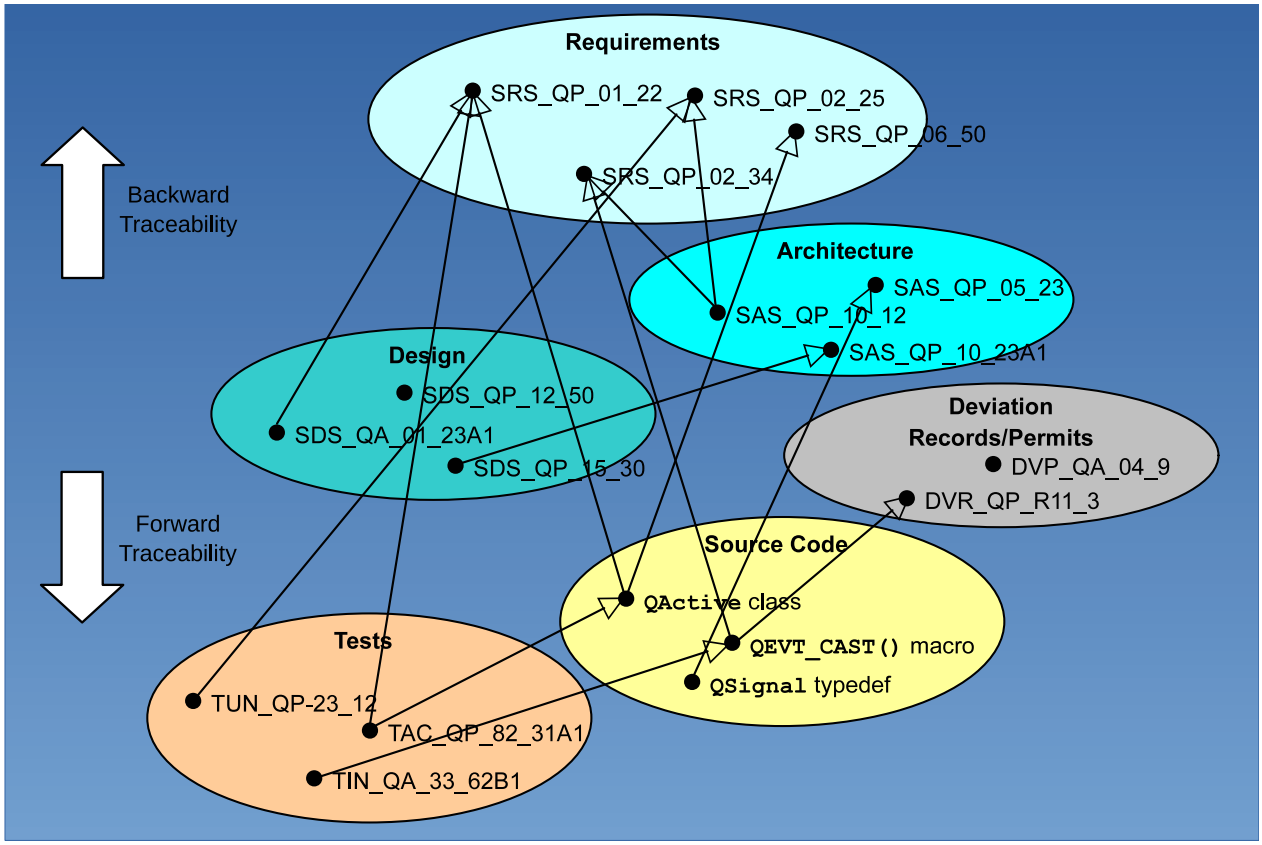
1.2.2 Backward Traceability

Backward traceability begins at a specific work artifact and links it to the original artifact. For example, architecture element can be linked with an upstream requirement, or code artifact with the upstream design. Backward traceability gives visibility into why specific artifacts were created and how different pieces of a system fit together. Tracing in this way also allows testers to find gaps or missing work artifacts.

Remarks

Backward traceability is the most natural and efficient way of specifying hierarchical relationships, such as superclass-subclass in object-oriented programming (OOP). Class inheritance is universally represented in the subclasses, which store their superclass (backward traceability). In contrast, superclasses don't show their subclasses (forward traceability). The *Spexygen* documentation applies this approach to all work artifacts, starting with the requirements at the top, through architecture, design, **source code**, tests, deviations, etc.

As illustrated in the diagram below, **backward traceability is provided explicitly** in the *Spexygen* documentation and the source code. Specifically, the downstream work artifacts provide trace information to the related upstream artifact by means of the Unique Identifier (UIDs).



Schematic View of Backward Traceability in the Spexygen documentation

Note

The *Spexygen* documentation traceability system includes the **source code**. This is achieved by placing special backward traceability links, such as `@ref SRS_PRJ_Foo_03 "SRS_PRJ_Foo_03"` or `@ref free_fun() "free_fun()"`, inside the *Spexygen* documentation for the source code.

1.2.3 Forward Traceability

Forward traceability begins at the original artifact and links it with all the resulting forward work items. For example, a requirement can be linked with source code elements that implement that requirement. This type of trace ensures that each original artifact (e.g., requirement) is not only satisfied but verified and validated. In the *Spexygen* documentation the forward traceability is **generated automatically** by the `spexygen.py` Python script.

Note

Forward traceability is typically **recursive** meaning that if artifact A traces to B and B traces to C, then artifact A also traces to C. *Spexygen* generates **recursive forward trace**, which enables the teams to perform *impact analysis* to identify the potential consequences of a change of a given artifact.

1.2.4 Bidirectional Traceability

Bidirectional traceability is the ability to perform both forward and backward traceability. Bidirectional traceability is the optimal type of traceability because it gives teams full visibility from requirements through architecture, design, source code, tests, and back again. The system implemented in the *Spexygen* documentation provides such bidirectional traceability.

Remarks

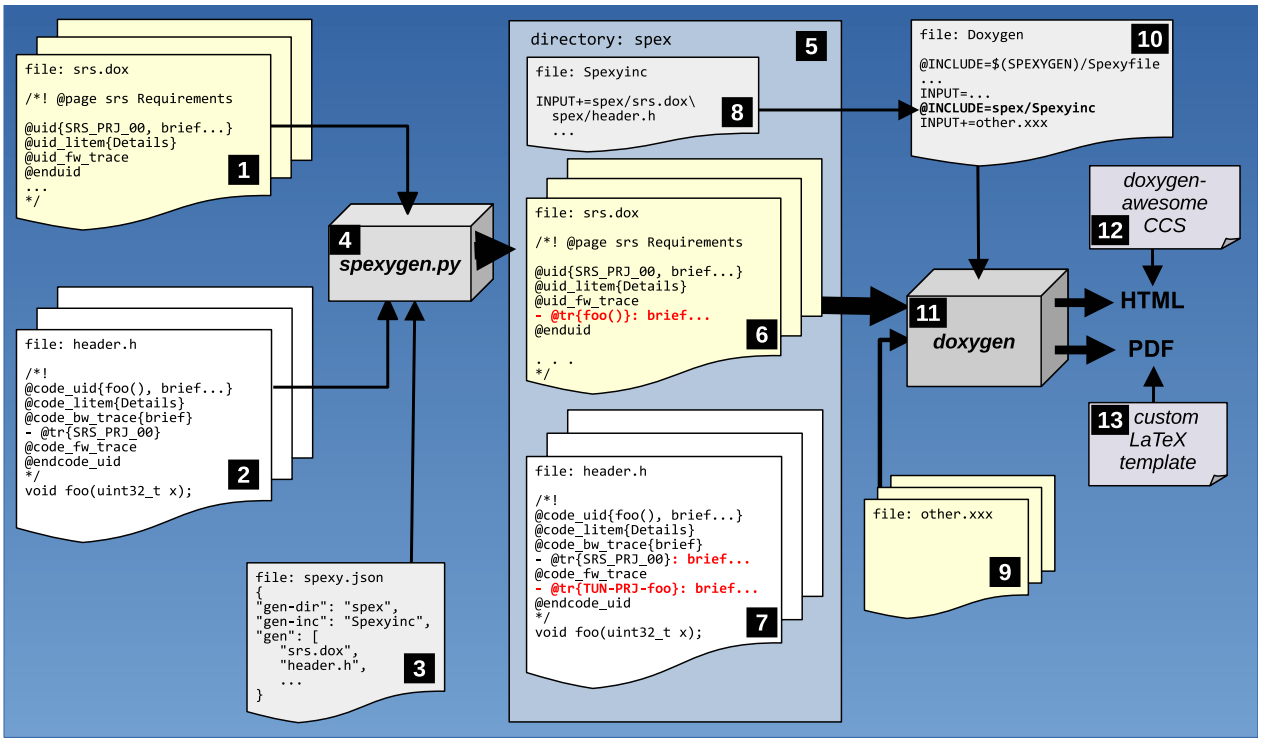
The whole system of traceability offered in *Spexygen* is **extensible** and can be used in any technical documentation system.

Chapter 2

Working with Spexygen

2.1 Spexygen Workflow

Spexygen can be viewed as a pre-processor for Doxygen↑ with the workflow similar to that of Doxygen. The diagram below shows the documentation generation steps, the relation between *Spexygen* and Doxygen, and the flow of information between them. The numbered sections following the diagram explain the labeled elements:



Spexygen information flow.

[1] Developers prepare documents (e.g., srs.dox) according to the conventions established by Doxygen. The individual "work artifacts" are created with a set of custom commands provided by *Spexygen* (file Spexyfile). For example, command @uid{} starts a definition of a new "work artifact" such as a requirement.

[2] Developers also apply the special custom commands provided by *Spexygen* to annotate the source code (e.g., header.h). For example, command @code_uid{} starts a definition of a new "code artifact".

[3] Developers prepare a *Spexygen* configuration file (e.g., spex.json in the diagram), which describes the documents to be traced and generated by *Spexygen*.

[4] The spexygen.py Python script processes the files and generates forward-traceabilty in requested locations in the provided files.

[5] The spexygen.py Python script creates a specified directory (spex in the diagram, see also "gen-dir": tag in spexy.json) to save the generated files.

[6-7] The spexygen.py Python script generates the augmented files into a specified directory. The generated files contain all the original information plus the tracing information generated by *Spexygen* (shown in red in the diagram)

[8] The spexygen.py Python script also generates the Spexyinc file that contains the information about all generated files.

[9] Developers can prepare other documentation, not processed by *Spexygen*, but also included in the final Doxygen-generated documentation.

[10] The Doxygen configuration file (Doxyfile) includes the Spexyfile (with the *Spexygen* custom commands) and the generated Spexyinc file (with files generated by *Spexygen* and to be processed by Doxygen)

[11] Doxygen processes the files generarted by *Spexygen* (see [6-7]) and other files (see [9]) according to the Doxyfile (see [10])

[12] If Doxygen is configured to produce HTML output, it applies the modern Doxygen-awesome HTML styling↑ (included in *Spexygen*)

[13] If Doxygen is configured to produce PDF output, it applies the modern LaTeX template provided by *Spexygen*

Note

The *Spexygen* workflow is illustrated in the provided example.

2.2 Preparing Spexygen Documentation

As described in the *spexygen* workflow, *Spexygen* works with the documentation and annotated code prepared according to the Doxygen conventions. However, *Spexygen* provides a layer of customized commands for defining traceable "work artifacts", which can be of two kinds:

1. work artifacts created in pure documentation, such as requirements specification; and
2. code artifacts created by Doxygen for various elements of source code, such as classes, functions, macros, etc.

2.2.1 Defining Work Artifacts

Work artifacts (e.g., requirements) are defined by means of the following *Spexygen* custom commands:

Command	Purpose
@uid{uid,brief}	starts the definition of a "work artifact" parameter: **uid** – the UID of the "work artifact" parameter: **brief** – brief description of the "work artifact"
@uid_litem{item}	adds new line - item in the "work artifact" definition parameter: **title** – Title of the line item(e.g., Details)
@uid_bw_trace{brief}	adds the backward trace section in in the "work artifact" definition parameter: **brief** – request *Spexygen* to add the brief item description
@uid_bw_trace	adds the backward trace section in in the "work artifact" definition overloaded version without requesting the brief description
@uid_fw_trace	adds the forward trace section in in the "work artifact" definition this is a request to *Spexygen* to **generate** the recursive forward traceability for the "work artifact"
@enduid	ends the definition "work artifact" must be placed at the end of "work artifact" definition
@tr{uid}	references the given UID parameter: **uid** – the UID of the "work artifact"

Remarks

The *Spexygen* custom commands are defined as `ALIAS=...` in `Spexyfile`.

The following snippet illustrates how a "work artifact" (a requirement) has been documented for *Spexygen* (see also file srs.dox):

```
[1] @section srs_req Requirements
    ...
[2] @uid{SRS_PRJ_Foo_03,My project class Foo shall provide a verify operation.}
[3] @uid_litem{Description}
    Longer description of the requirement
[4] @uid_bw_trace{brief}
[5] - @tr{SRS_PRJ_Foo_00}
[6] @uid_fw_trace
[7] @enduid
```

`[1]` *Spexygen* "work artifacts" must be defined in the scope of a Doxygen **@section**.

`[2]` each "work artifact" is defined with the *Spexygen* command **@uid{}**, which takes two arguments:

1. the UID associated with the artifact (e.g., `SRS_PRJ_Foo_03`)
2. the brief description of the artifact (e.g., `My project class Foo shall provide a verify operation.`)

Attention

The whole **@uid{}** command must be defined in a single line of text and the brief description must not contain commas **,**

`[3]` the "work artifact" definition can contain "line items", such as "Description" coded by means of the **@uid_litem{Description}** *Spexygen* command. A "work artifact" can have multiple line items defined with the **@uid_litem{}** *Spexygen* command.

`[4]` the "work artifact" can specify backward traceability by means of the **@uid_bw_trace** *Spexygen* command. If this command provides argument `{brief}`, *Spexygen* will generate the brief description for each of the provided traceability links (see the next step `[5]`)

`[5]` the traceability links to the upstream artifacts must be provided explicitly by means of the **@tr{uid}** *Spexygen* command. The command establishes traceability to the UID provided in the argument (e.g., `SRS_PRJ_Foo_00`)

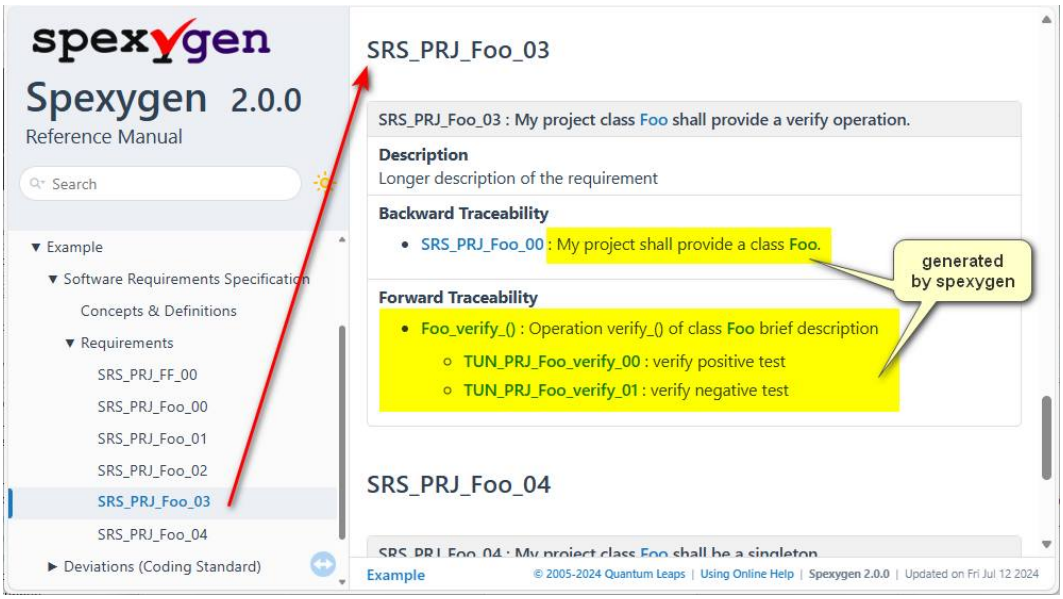
`[6]` the "work artifact" can specify forward traceability by means of the **@uid_fw_trace** *Spexygen* command

Note

The **@uid_fw_trace** *Spexygen* command is just a **placeholder** for *Spexygen* to generate the forward traceability links at this place.

`[7]` The artifact definition must end with the *Spexygen* **@enduid** command

The following screen shot shows how the "work artifact" defined in the code snipped above is rendered in HTML:



Examples of "work artifacts" (requirements)

2.2.2 Defining Code Artifacts

Code artifacts (e.g., functions, macros, classes) are naturally handled by Doxygen and the *Spexygen* system must comply with the exiting Doxygen conventions. The "code artifacts" are defined by means of the following *Spexygen* custom commands:

Command	Purpose
@code_uid{uid,brief }	starts the definition of a "code artifact" parameter: **uid** –the UID of the "code artifact" parameter: **brief** – brief description of the "code artifact"
@code_litem{item }	adds new line-item in the "code artifact" definition parameter: **title** –Title of the line item(e.g., Details)
@code_bw_trace{brief}	adds the backward trace section in in the "code artifact" definition parameter: **brief** – request *Spexygen* to add the brief item description
@code_bw_trace	adds the backward trace section in in the "code artifact" definition overloaded version without requesting the brief description
@code_fw_trace	adds the forward trace section in in the "code artifact" definition this is a request to *Spexygen* to **generate** the recursive forward traceability for the "code artifact"
@endcode_uid	ends the definition "code artifact" must be placed at the end of "code artifact" definition
@tr{uid}	references the given UID parameter: **uid** – the UID of the "code artifact"

Remarks

The *Spexygen* custom commands are defined as `ALIAS=...` in `Spexyfile`.

The following snippet illustrates how a "code artifact" (function `free_fun()`) has been documented (see also file header.h):

```
    /*!
[1] @code_uid{Foo_verify_(), Operation verify_() of class Foo brief description}
[2] @code_litem{Details}
    Operation verify_() of class Foo longer description.

    @param[in]  me - the instance pointer (OOP in C)

[3] @code_bw_trace{brief}
[4] - @tr{SRS_PRJ_Foo_03}
[5] @code_fw_trace
[6] @endcode_uid
    */
[7] bool Foo_verify_(Foo const* const me);
```

`[1]` each "code artifact" is defined with the *Spexygen* command **@code_uid{}**, which takes two arguments :

1. the UID associated with the artifact(e.g., `Foo_verify_()`)
2. the brief description of the artifact(e.g., `Operation verify_() of class Foo brief description`)

Attention

The whole **@code_uid{}** command must be defined in a single line of text and the brief description must not contain commas **,**

`[2]` the "code artifact" definition can contain "line items", such as "Details" coded by means of the **@code_litem{Details}** *Spexygen* command. A "code artifact" can have multiple items defined with the **@code_litem{...}** *Spexygen* command.

`[3]` the "code artifact" can specify backward traceability by means of the **@uid_bw_trace** *Spexygen* command. If this command provides argument `{brief}`, *Spexygen* will generate the brief description for each of the provided traceability links(see the next step `[4]`)

`[4]` the traceability links to the upstream artifacts must be provided explicitly by means of the **@tr{uid}** *Spexygen* command. The command establishes traceability to the UID provided in the argument(e.g., `SRS_PRJ_Foo_03`)

`[5]` the "code artifact" can specify forward traceability by means of the **@code_fw_trace** *Spexygen* command

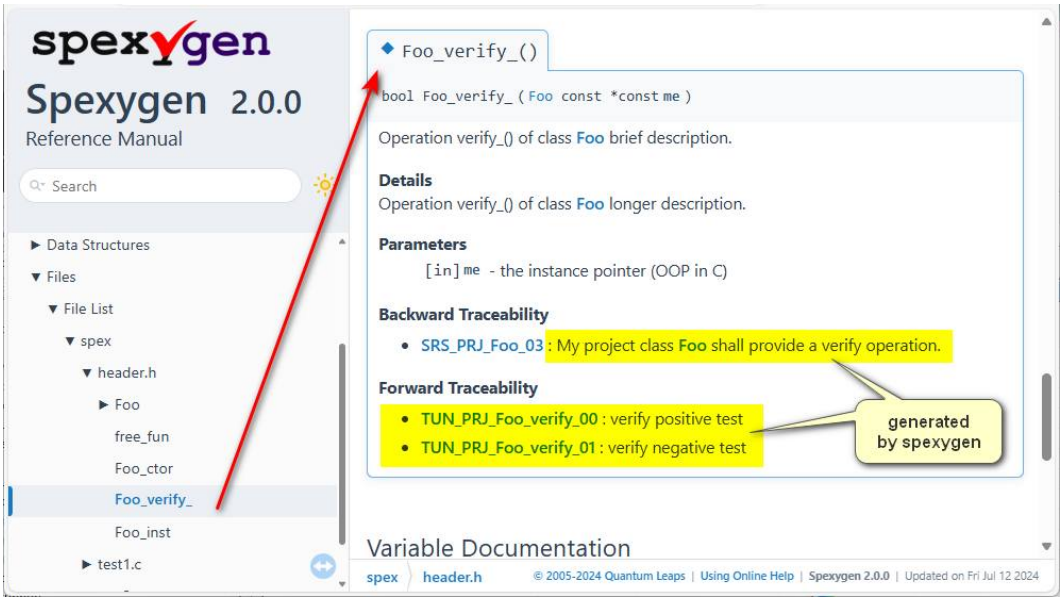
Note

The **@code_fw_trace** *Spexygen* command is just a **placeholder** for *Spexygen* to generate the forward traceability links at this place.

`[6]` The "code artifact" definition must end with the *Spexygen* **@endcode_uid** command

`[7]` Finally, the "code artifact" needs to be declared, so that Doxygen can analyze the syntax of the specific programming language.

The following screen shot shows how the "code artifact" defined in the code snipped above is rendered in HTML :



Examples of "code artifacts" (function)

2.3 Generating Spexygen Documentation

As shown in the spexygen workflow diagram, *Spexygen* documentation generation is handled by the Python script `spexygen.py`. This script processes the input files in two passes. In the first pass (called "trace"), the `spexygen.py` script parses the files for the special *Spexygen* commands and collects the information about the dependencies among the various artifacts (based on their UIDs). In the second pass (called "gen"), the `spexygen.py` script generates the output-files by replacing the special forward traceability commands (**@ui_fw_trace** and **@code_fw_trace**) with the actual traceability links collected during the first pass.

2.3.1 Spexygen Configuration File

Similar to Doxygen, *Spexygen* is configured by an external configuration file with the default name `spex.json`. That file specifies about the input files and output directory and files. The file is structure according to the JSON format↑. Below is an annotated example of `spex.json` file:

```
    {
[1] "trace": [
        "../example/exa.dox",
        "../example/srs.dox",
        "../example/dev.dox",
        "../example/inc",
        "../example/src",
        "../example/test"
    ]

[2] "gen-dir": "spex",
[3] "gen-inc": "Spexyinc",
[4] "gen": [
        "../example/exa.dox",
        "../example/srs.dox",
        "../example/dev.dox",
        "../example/inc",
        "../example/src",
        "../example/test"
    ],
    }
```

`[1]` The `"trace":` JSON array contains a list of the input files that *Spexygen* will "trace".

Remarks

The `"trace":` array is optional and if it is not provided, *Spexygen* will "trace" the files specified in the `"gen":` array.

`[2]` The `"gen-dir":` JSON string specifies the output-directory into which *Spexygen* will generate the output

Remarks

The "gen-dir":`JSON string is optional and if not provided, _Spexygen_ will use the default directory`spex`

`[3]` The "gen-inc": JSON string specifies the spexygen include-file for Doxygen, which contains the list of the generated files. It is intended for inclusion in the `Doxyfile`. The spexygen include-file is generated in the output-directory.

Remarks

The "gen-inc":` JSON string is optional and if not provided, *Spexygen* will NOT generate the include-file.

`[4]` The `"gen":` JSON array contains a list of the files that *Spexygen* will generate by replacing the special forward traceability commands (**@ui_fw_trace** and **@code_fw_trace**) with the actual traceability links collected during the "trace" pass.

2.3.2 Running Spexygen

The `spexygen.py` Python script can be executed from a command prompt. Typically, you run the script in the directory, where you have your `spex.json` script.Here is an example run:

```
cd spexygen/doc  <-- the directory with the spex.json config file
python ../spexygen.py

Spexygen: traceable technical documentation system 2.0.0
Copyright (c) 2005-2024 Quantum Leaps, www.state-machine.com

Tracing: ../example/exa.dox
Tracing: ../example/srs.dox
Tracing: ../example/dev.dox
Tracing: ../example/inc/header.h
Tracing: ../example/src/source.c
Tracing: ../example/test/test1.c
Tracing: ../example/test/test2.c
Generating: spex/exa.dox
Generating: spex/srs.dox
Generating: spex/dev.dox
Generating: spex/header.h
  No forward trace for UID: "Foo"
  No forward trace for UID: "Foo_inst"
Generating: spex/source.c
Generating: spex/test1.c
Generating: spex/test2.c
```

Note

The `spexygen.py` script takes one command-line parameter, which is the name of the configuration file. If not provided, the default name is `spex.json`.

Remarks

The `spexygen.py` Python script is also available in the `PyPi package manager↑` and can be installed with the standard Python package installer `pip`:

```
pip install spexygen
```

After such installation, you run *Spexygen* simply as follows:

```
spexygen
```

Either way, depending on your settings *Spexygen* will create the output-directory (e.g., `spex`) with all the generated files.

2.3.2.1 Spexygen Include File

Spexygen can generate the spexygen include-file for Doxygen, which contains the list of the generated files. It is intended for inclusion in the `Doxyfile`. The spexygen include-file is generated in the output-directory. Here is an example:

```
INPUT += \
spex/exa.dox \
spex/srs.dox \
spex/dev.dox \
spex/header.h \
spex/source.c \
spex/test1.c \
spex/test2.c
```

2.3.3 Doxygen Configuration File

The *Spexygen* output (plus other files) can be now fed to Doxygen. However, before you can run Doxygen, you need the `Doxygen configuration file`↑. That configuration file must include the spexygen special commands and the files generated by *Spexygen*. Here is an example of an annotated `Doxyfile` with these elements:

```
[1] @INCLUDE = $(SPEXYGEN)/Spexyfile
[2] INPUT = main.dox
[3] @INCLUDE = spex/Spexyinc
[4] INPUT += ...
```

`[1]` The Doxygen `\@INCLUDE` tag includes the `Spexyfile` located in the `$(SPEXYGEN)` directory defined here by means of an environment variable.

Attention

> The `SPEXYGEN` environment variable must be defined in your system to point to the *spexgyen* installation directory.

`[2]` The `INPUT tag`↑ specifies the input files for Doxygen. Here you can specify files that you wish to include in the Doxygen output, but which have not been processed by *Spexygen*.

`[3]` This Doxygen `\@INCLUDE = spex/Spexyinc` tag includes the spexgyen-include file with the Doxygen input generated by *Spexygen*.

`[4]` Any additional Doxygen `INPUT` (not produced by *spexyen*) can be specified as well (please note the `+=` operator as opposed to `=`)

2.3.4 Running Doxygen

Once the `$(SPEXYGEN)` environment variable has been defined, Doxygen can be run as usual from the directolry with the `Doxyfile`:

```
Doxygen
```

2.3.4.1 Combining Spexygen & Doxygen

In practice, most convenient is combining *Spexygen* and Doxygen and run both automatically one after another. Here is an example Windows batch file that automates the process (see `spexygen/doc/make.bat`):

Note

The provided `make.bat` can generate HTML↑ and PDF↑ output formats.

```
@setlocal

@echo usage:
@echo make
@echo make -PDF

:: tools (adjust to your system)-------------------------------------------
:: Doxygen/Spexygen tools
@set DOXYGEN=Doxygen
@set SPEXYGEN=..

@echo Generate Spexygen tracing ------------------------------------------
rmdir /S /Q  .\spex
python %SPEXYGEN%/spexygen.py spex.json

::============================================================================
@if "%1"=="-PDF" goto PDF

@echo Generate HTML Documentation ----------------------------------------
@set HTML_OUT=html

@echo.
@echo cleanup
rmdir /S /Q %HTML_OUT%

@echo generating HTML...
%DOXYGEN% Doxyfile

@echo Adding custom files...
copy %SPEXYGEN%\spexygen-awesome\jquery.js %HTML_OUT%

::qclean %HTML_OUT%
goto END

:PDF
@echo Generate PDF Documentation -----------------------------------------
@set LATEX_OUT=latex

@echo.
@echo cleanup
rmdir /S /Q  %LATEX_OUT%

@echo generating LATEX...
%DOXYGEN% Doxyfile-PDF

:: Generate LaTex/PDF Documentation...
@echo generating PDF...
@cd %LATEX_OUT%
@call make.bat
@copy refman.pdf ..\DOC-MAN-SPX.pdf
@cd ..
rmdir /S /Q  %LATEX_OUT%

:END
@echo Final cleanup ------------------------------------------------------
rmdir /S /Q  .\spex

@endlocal
```

Chapter 3

Example

3.1 About this Example

Simple example for Doxygen.

```
+---example/     <-- this example
|   |   main.dox <-- example description
|   |   srs.dox  <-- example Software Requirements Specification
|   |
|   +---inc
|   |       header.h <-- example header file
|   |
|   +---src
|   |       source.c <-- example source file
|   |
|   \---test
|           test1.c  <-- example test file
|           test2.c  <-- example test file
```

3.2 Spexygen-Generated Documentation

Figure 3.1 image caption

The following sections contain the *Spexygen*-generated documentation of this example:

- Software Requirements Specification

- header.h
- source.c
- test1.c
- test2.c

Note

Please note the generated *forward-traceability* and the *backward-traceability* links (augmented with the brief descriptions) in the "work artifacts" and "code artifacts". Also, try clicking on the provided traceability links as well as *searching* the UIDs in the Doxygen "search" box.

3.3 Software Requirements Specification

This is an example Software Requirements Specification (SRS), illustrating the typical structure and the use of *Spexygen* commands to define the **traceable** requirement work artifacts.

Note

Please note the generated *forward-traceability* and the *backward-traceability* links (augmented with the brief descriptions) in the "work artifacts" and "code artifacts". Also, try clicking on the provided traceability links as well as *searching* the UIDs in the Doxygen "search" box.

3.3.1 Concepts & Definitions

Description of concepts and definitions...

3.3.2 Requirements

Definitions of formal requirements specifications with *Spexygen* commands.

3.3.2.1 SRS_EXA_FF_00

SRS_EXA_FF_00 : My project shall provide a free function foo().
Description Longer description of the requirement
Forward Traceability • free_fun() : Free function brief description – TUN_PRJ_free_fun_00 : zero input test – TUN_PRJ_free_fun_01 : non-zero input test

3.3.2.2 SRS_EXA_Foo_00

SRS_EXA_Foo_00 : My project shall provide a class Foo.

Description
Longer description of the requirement

Forward Traceability

- SRS_EXA_Foo_01 : Class Foo shall provide a public attribute x.
  - Foo::x : Attribute x of class Foo brief description
    - Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x
- SRS_EXA_Foo_02 : Class Foo shall provide a constructor.
  - Foo_ctor() : Constructor of class Foo brief description
    - TUN_PRJ_Foo_ctor_01 : constructor test
- SRS_EXA_Foo_03 : Class Foo shall provide a verify operation.
  - Foo_verify_() : Verify operation to check the class invariant
    - Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x
    - TUN_PRJ_Foo_verify_00 : verify positive test
    - TUN_PRJ_Foo_verify_01 : verify negative test
- SRS_EXA_Foo_04 : Class Foo shall provide an update operation.
  - Foo_update_() : Update operation to update the class invariant
- SRS_EXA_Foo_05 : Class Foo shall be a singleton.
  - Foo_inst : Foo instance brief description (singleton)
- Foo : Class Foo brief description
  - Foo_update_() : Update operation to update the class invariant
  - Foo_inst : Foo instance brief description (singleton)

3.3.2.3 SRS_EXA_Foo_01

SRS_EXA_Foo_01 : Class Foo shall provide a public attribute x.

Description
Longer description of the requirement

Backward Traceability

- SRS_EXA_Foo_00 : My project shall provide a class Foo.

Forward Traceability

- Foo::x : Attribute x of class Foo brief description
  - Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x

3.3.2.4 SRS_EXA_Foo_02

SRS_EXA_Foo_02 : Class Foo shall provide a constructor.
Description Longer description of the requirement
Backward Traceability • SRS_EXA_Foo_00 : My project shall provide a class Foo.
Forward Traceability • Foo_ctor() : Constructor of class Foo brief description – TUN_PRJ_Foo_ctor_01 : constructor test

3.3.2.5 SRS_EXA_Foo_03

SRS_EXA_Foo_03 : Class Foo shall provide a verify operation.
Description Longer description of the requirement
Backward Traceability • SRS_EXA_Foo_00 : My project shall provide a class Foo.
Forward Traceability • Foo_verify_() : Verify operation to check the class invariant – Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x – TUN_PRJ_Foo_verify_00 : verify positive test – TUN_PRJ_Foo_verify_01 : verify negative test

3.3.2.6 SRS_EXA_Foo_04

SRS_EXA_Foo_04 : Class Foo shall provide an update operation.
Description Longer description of the requirement
Backward Traceability • SRS_EXA_Foo_00 : My project shall provide a class Foo.

Forward Traceability

- Foo_update_() : Update operation to update the class invariant

3.3.2.7 SRS_EXA_Foo_05

SRS_EXA_Foo_05 : Class Foo shall be a singleton.
Description Longer description of the requirement
Backward Traceability • SRS_EXA_Foo_00 : My project shall provide a class Foo.
Forward Traceability • Foo_inst : Foo instance brief description (singleton)

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Data Structure Documentation

6.1 Foo Struct Reference

Class Foo brief description.

`#include "header.h"`

Data Fields

- uint32_t x

  Attribute x of class Foo brief description.
- uint32_t x_dis

  Duplicate Inverse Storage for attribute Foo::x.

6.1.1 Detailed Description

Class Foo brief description.

Details

Class Foo longer description.

Backward Traceability

- SRS_EXA_Foo_00 : My project shall provide a class Foo.

Forward Traceability

- Foo_ctor() : Constructor of class Foo brief description
  - TUN_PRJ_Foo_ctor_01 : constructor test
- Foo_verify_() : Verify operation to check the class invariant
  - Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x
  - TUN_PRJ_Foo_verify_00 : verify positive test
  - TUN_PRJ_Foo_verify_01 : verify negative test
- Foo_update_() : Update operation to update the class invariant
- Foo_inst : Foo instance brief description (singleton)

6.1.2 Field Documentation

6.1.2.1 x

```
uint32_t Foo::x
```

Attribute x of class Foo brief description.

Details

Attribute x of class Foo: longer description.

Backward Traceability

- SRS_EXA_Foo_01 : Class Foo shall provide a public attribute x.

6.1.2.2 x_dis

```
uint32_t Foo::x_dis
```

Duplicate Inverse Storage for attribute Foo::x.

Details

Duplicate Inverse Storage (DIS) for attribute Foo::x: longer description.

Backward Traceability

- Foo::x : Attribute x of class Foo brief description
- Foo_verify_() : Verify operation to check the class invariant

The documentation for this struct was generated from the following file:

- header.h

Chapter 7

File Documentation

7.1 main.dox File Reference

7.2 main.dox File Reference

7.3 header.h File Reference

```
#include <stdint.h>
#include <stdbool.h>
```

Data Structures

- struct Foo

  Class Foo brief description.

Functions

- uint8_t const * free_fun (uint32_t x)

  Free function brief description.
- void Foo_ctor (Foo *const me, uint32_t const x)

  Constructor of class Foo brief description.
- bool Foo_verify_ (Foo const *const me)

  Verify operation to check the class invariant.
- void Foo_update_ (Foo *const me)

  Update operation to update the class invariant.

Variables

- Foo const Foo_inst

  Foo instance brief description (singleton)

7.3.1 Function Documentation

7.3.1.1 free_fun()

```
uint8_t const * free_fun (
            uint32_t x)
```

Free function brief description.

Details

Free function longer description.

Parameters

in	*x*	- the parameter x

Returns

pointer to a static array

Backward Traceability

- SRS_EXA_FF_00 : My project shall provide a free function foo().

Forward Traceability

- TUN_PRJ_free_fun_00 : zero input test
- TUN_PRJ_free_fun_01 : non-zero input test

7.3.1.2 Foo_ctor()

```
void Foo_ctor (
            Foo *const me,
            uint32_t const x)
```

Constructor of class Foo brief description.

Details

Constructor of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)
in	*x*	- the initial value for me->x

Backward Traceability

- SRS_EXA_Foo_02 : Class Foo shall provide a constructor.
- Foo : Class Foo brief description

Forward Traceability

- TUN_PRJ_Foo_ctor_01 : constructor test

7.3.1.3 Foo_verify_()

```
bool Foo_verify_ (
             Foo const *const me)
```

Verify operation to check the class invariant.

Details

Operation verify_() of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)

Returns

'true' when the Foo instance verification succeeds, 'false' otherwise.

Backward Traceability

- SRS_EXA_Foo_03 : Class Foo shall provide a verify operation.
- Foo : Class Foo brief description

Forward Traceability

- Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x
- TUN_PRJ_Foo_verify_00 : verify positive test
- TUN_PRJ_Foo_verify_01 : verify negative test

7.3.1.4 Foo_update_()

```
void Foo_update_ (
             Foo *const me)
```

Update operation to update the class invariant.

Details

Constructor of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)

Backward Traceability

- SRS_EXA_Foo_04 : Class Foo shall provide an update operation.
- Foo : Class Foo brief description

Forward Traceability

7.3.2 Variable Documentation

7.3.2.1 Foo_inst

```
Foo const Foo_inst  [extern]
```

Foo instance brief description (singleton)

Details

Foo instance longer description.

Backward Traceability

- SRS_EXA_Foo_05 : Class Foo shall be a singleton.
- Foo : Class Foo brief description

Forward Traceability

7.4 source.c File Reference

```
#include "header.h"
```

Functions

- uint8_t const * free_fun (uint32_t x)

  Free function brief description.
- void Foo_ctor (Foo *const me, uint32_t const x)

  Constructor of class Foo brief description.
- void Foo_update_ (Foo *const me)

  Update operation to update the class invariant.
- bool Foo_verify_ (Foo const *const me)

  Verify operation to check the class invariant.

7.4.1 Function Documentation

7.4.1.1 free_fun()

```
uint8_t const * free_fun (
            uint32_t x)
```

Free function brief description.

Details

Free function longer description.

Parameters

in	*x*	- the parameter x

Returns

pointer to a static array

Backward Traceability

- SRS_EXA_FF_00 : My project shall provide a free function foo().

Forward Traceability

- TUN_PRJ_free_fun_00 : zero input test
- TUN_PRJ_free_fun_01 : non-zero input test

7.4.1.2 Foo_ctor()

```
void Foo_ctor (
            Foo *const me,
            uint32_t const x)
```

Constructor of class Foo brief description.

Details

Constructor of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)
in	*x*	- the initial value for me->x

Backward Traceability

- SRS_EXA_Foo_02 : Class Foo shall provide a constructor.
- Foo : Class Foo brief description

Forward Traceability

- TUN_PRJ_Foo_ctor_01 : constructor test

7.4.1.3 Foo_update_()

```
void Foo_update_ (
            Foo *const me)
```

Update operation to update the class invariant.

Details

Constructor of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)

Backward Traceability

- SRS_EXA_Foo_04 : Class Foo shall provide an update operation.
- Foo : Class Foo brief description

Forward Traceability

7.4.1.4 Foo_verify_()

```
bool Foo_verify_ (
            Foo const *const me)
```

Verify operation to check the class invariant.

Details

Operation verify_() of class Foo longer description.

Parameters

in	*me*	- the instance pointer (OOP in C)

Returns

'true' when the Foo instance verification succeeds, 'false' otherwise.

Backward Traceability

- SRS_EXA_Foo_03 : Class Foo shall provide a verify operation.
- Foo : Class Foo brief description

Forward Traceability

- Foo::x_dis : Duplicate Inverse Storage for attribute Foo::x
- TUN_PRJ_Foo_verify_00 : verify positive test
- TUN_PRJ_Foo_verify_01 : verify negative test

7.5 srs.dox File Reference

7.6 test1.c File Reference

```
#include "header.h"
#include "unity.h"
```

Functions

- void setUp (void)
- void tearDown (void)
- void TUN_PRJ_free_fun_00 (void)

  zero input test
- void TUN_PRJ_free_fun_01 (void)

  non-zero input test

7.6.1 Function Documentation

7.6.1.1 setUp()

```
void setUp (
            void )
```

7.6.1.2 tearDown()

```
void tearDown (
             void )
```

7.6.1.3 TUN_PRJ_free_fun_00()

```
void TUN_PRJ_free_fun_00 (
             void )
```

zero input test

Details

This test checks that zero input to free_fun() produces zero array.

Backward Traceability

- free_fun() : Free function brief description

7.6.1.4 TUN_PRJ_free_fun_01()

```
void TUN_PRJ_free_fun_01 (
             void )
```

non-zero input test

Details

This test checks that non-zero input to free_fun() produces expected array.

Backward Traceability

- free_fun()

7.7 test2.c File Reference

```
#include "header.h"
#include "unity.h"
```

Functions

- void setUp (void)
- void tearDown (void)
- void TUN_PRJ_Foo_ctor_01 (void)

  constructor test
- void TUN_PRJ_Foo_verify_00 (void)

  verify positive test
- void TUN_PRJ_Foo_verify_01 (void)

  verify negative test

7.7.1 Function Documentation

7.7.1.1 setUp()

```
void setUp (
            void )
```

7.7.1.2 tearDown()

```
void tearDown (
            void )
```

7.7.1.3 TUN_PRJ_Foo_ctor_01()

```
void TUN_PRJ_Foo_ctor_01 (
            void )
```

constructor test

Details

This test checks that Foo_ctor() produces valid instance.

Backward Traceability

- Foo_ctor() : Constructor of class Foo brief description

7.7.1.4 TUN_PRJ_Foo_verify_00()

```
void TUN_PRJ_Foo_verify_00 (
             void )
```

verify positive test

Details

This test checks that Foo_verify_() distinguishes valid instance.

Backward Traceability

- Foo_verify_() : Verify operation to check the class invariant

7.7.1.5 TUN_PRJ_Foo_verify_01()

```
void TUN_PRJ_Foo_verify_01 (
             void )
```

verify negative test

Details

This test checks that Foo_verify_() distinguishes invalid instance.

Backward Traceability

- Foo_verify_() : Verify operation to check the class invariant

7.8 help.dox File Reference

Index