
# RTEMS Software Engineering

*Release 7.0-rc1 (24th January 2026)*

# CONTENTS

The RTEMS Project is hosted at https://www.rtems.org. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

**RTEMS Online Resources**

- Home https://www.rtems.org
- Documentation https://docs.rtems.org
- Mailing Lists https://lists.rtems.org
- Bug Reporting https://gitlab.rtems.org
- Git Repositories https://gitlab.rtems.org
- Developers https://gitlab.rtems.org

CHAPTER

**ONE**

# PREFACE

This manual aims to guide the development of RTEMS itself. You should read this document if you want to participate in the development of RTEMS. Users of RTEMS may find background information in this manual. Please refer to the RTEMS User Manual and RTEMS Classic API Guide if you want to know how the RTEMS development environment is set up and how you can develop applications using RTEMS.

CHAPTER

**TWO**

# RTEMS PROJECT MISSION STATEMENT

RTEMS development done under the umbrella of the RTEMS Project aims to provide a free and open real-time operating system targeted towards deeply embedded systems which is competitive with proprietary products. The RTEMS Project encourages the support and use of standard APIs in order to promote application portability and ease porting other packages to the RTEMS environment.

The RTEMS development effort uses an open development environment in which all users collaborate to improve RTEMS. The RTEMS cross development tool suite is based upon the free GNU tools and the open source standard C library newlib. RTEMS supports many host platforms and target architectures.

## 2.1 Free Software Project

The free software goals of the project are:

- RTEMS and supporting components are available under various free licenses with copyrights being held by individual authors.
- All software which executes on the target will not place undue restrictions on embedded applications. See also *Licensing Requirements* (page 279).
- Patches must be legally acceptable for inclusion into the RTEMS Project or the specific project being used.

## 2.2 Design and Development Goals

- Source based development with all users building from source
- Any suitable host should be supported
- Open testing, tests and test results
- Ports to new architectures and CPU models
- Addition of Board Support Packages for available hardware
- Improved runtime libraries
- Faster debug cycle
- Various other infrastructure improvements

## 2.3 Open Development Environment

- Encourage cooperation and communication between developers
- Work more closely with “consumers”
- Code available to everyone at any time, and everyone is welcome to participate in development
- Patches will be considered equally based on their technical merits
- All individuals and companies are welcome to contribute as long as they accept the ground rules
- Open mailing lists
- Developer friendly tools and procedures with a focus on keeping them current
- Conflicts of interest exist for many RTEMS developers. The developers contributing to the RTEMS Project must put the interests of the RTEMS Project first.

CHAPTER

**THREE**

# RTEMS STAKEHOLDERS

You are a potential RTEMS stakeholder. RTEMS is a community based free and open source project. All users are treated as stakeholders. It is hoped that as stakeholders, users will contribute to the project, sponsor core developers, and help fund the infrastructure required to host and manage the project. Please have a look at the *Support and Contributing* chapter of the RTEMS User Manual.

CHAPTER

**FOUR**

# INTRODUCTION TO PRE-QUALIFICATION

RTEMS has a long history of being used to support critical applications. In some of these application domains, there are standards (e.g., DO-178C, NPR 7150.2) which define the expectations for the processes used to develop software and the associated artifacts. These standards typically do not specify software functionality but address topics like requirements definition, traceability, having a documented change process, coding style, testing requirements, and a user's manual. During system test, these standards call for a review - usually by an independent entity - that the standard has been adhered to. These reviews cover a broad variety of topics and activities, but the process is generally referred to as qualification, verification, or auditing against the specific standard in use. The RTEMS Project will use the term "qualification" independent of the standard.

The goal of the RTEMS Qualification Project is to make RTEMS easier to review regardless of the standard chosen. Quite specifically, the RTEMS Qualification effort will NOT produce a directly qualified product or artifacts in the format dictated by a specific organization or standard. The goal is to make RTEMS itself, documentation, testing infrastructure, etc. more closely align with the information requirements of these high integrity qualification standards. In addition to improving the items that a mature, high quality open source project will have, there are additional artifacts needed for a qualification effort that no known open source project possesses. Specifically, requirements and the associated traceability to source code, tests, and documentation are needed.

The RTEMS Qualification Project is technically "pre-qualification." True qualification must be performed on the project's target hardware in a system context. The FAA has provided guidance for Reusable Software Components (FAA-AC20-148) and this effort should follow that guidance. The open RTEMS Project, with the assistance of domain experts, will possess and maintain the master technical information needed in a qualification effort. Consultants will provide the services required to tailor the master information, perform testing on specific system hardware, and to guide end users in using the master technical data in the context of a particular standard.

The RTEMS Qualification Project will broadly address two areas. The first area is suggesting areas of improvement for automated project infrastructure and the master technical data that has traditionally been provided by the RTEMS Project. For example, the RTEMS Qualification could suggest specific improvements to code coverage reports. The teams focused on qualification should be able to provide resources for improving the automated project infrastructure and master technical data for RTEMS. The term "resources" is often used by open source projects to refer to volunteer code contributions or funding. Although code contributions in this area are important and always welcome, funding is also important. At a minimum, ongoing funding is

needed for maintenance and upgrades of the RTEMS Project server infrastructure, addition of services to those servers, and core contributors to review submissions

The second area is the creation and maintenance of master technical data that has traditionally not been owned or maintained by the RTEMS Project. The most obvious example of this is a requirements set with proper infrastructure for tracing requirements through code to test and documentation. It is expected that these will be maintained by the RTEMS Qualification Project. They will be evaluated for adoption by the main RTEMS Project but the additional maintenance burden imposed will be a strong factor in this consideration. It behooves the RTEMS Qualification Project to limit dependence on manual checks and ensure that automation and ongoing support for that automation is contributed to the RTEMS Project.

It is expected that the RTEMS Qualification Project will create and maintain maps from the RTEMS master technical data to the various qualification standards. It will maintain "score-cards" which identify how the RTEMS Project is currently doing when reviewed per each standard. These will be maintained in the open as community resources which will guide the community in improving its infrastructure.

## 4.1 Stakeholder Involvement

Qualification of RTEMS is a specialized activity and only specific users of RTEMS will complete a formal qualification activity. The RTEMS Project cannot self-fund this entire activity and requires stakeholders to invest on an ongoing basis to ensure that any investment they make is maintained and viable in the long-term. The RTEMS core developers view steady support of the qualification effort as necessary to continue to lower the overall costs of qualifying RTEMS.

CHAPTER

**FIVE**

# SOFTWARE REQUIREMENTS ENGINEERING

Software engineering standards for critical software such as ECSS-E-ST-40C demand that software requirements for a software product are collected in a software requirements specification (technical specification in ECSS-E-ST-40C terms). They are usually derived from system requirements (requirements baseline in ECSS-E-ST-40C terms). RTEMS is designed as a reusable software product which can be utilized by application designers to ease the development of their applications. The requirements of the end system (system requirements) using RTEMS are only known to the application designer. RTEMS itself is developed by the RTEMS maintainers and they do not know the requirements of a particular end system in general. RTEMS is designed as a real-time operating system to meet typical system requirements for a wide range of applications. Its suitability for a particular application must be determined by the application designer based on the technical specification provided by RTEMS accompanied with performance data for a particular target platform.

Currently, no technical specification of RTEMS exists in the form of a dedicated document. Since the beginning of the RTEMS evolution in the late 1980s it was developed iteratively. It was never developed in a waterfall model. During initial development the RTEID [Mot88] and later the ORKID [VIT90] draft specifications were used as requirements. These were evolving during the development and an iterative approach was followed often using simple algorithms and coming back to optimise. In 1993 and 1994 a subset of pthreads sufficient to support *GNAT* was added as requirements. At this time the Ada tasking was defined, however, not implemented in GNAT, so this involved guessing during the development. Later some adjustments were made when Ada tasking was actually implemented. So, it was consciously iterative with the specifications evolving and feedback from performance analysis. Benchmarks published from other real time operating systems were used for comparison. Optimizations were carried out until the results were comparable. Development was done with distinct contractual phases and tasks for development, optimization, and the addition of priority inheritance and rate monotonic scheduling. The pthreads requirement has grown to be as much POSIX as possible.

Portability from FreeBSD to use its network stack, USB stack, SD/MMC card stack and device drivers resulted in another set of requirements. The addition of support for symmetric multiprocessing (SMP) was a huge driver for change. It was developed step by step and sponsored by several independent users with completely different applications and target platforms in mind. The high performance OpenMP support introduced the Futex as a new synchronization primitive.

**Guidance**

A key success element of RTEMS is the ability to accept changes driven by user needs and still keep the operating system stable enough for production systems. Procedures that place a high burden on changes are doomed to be discarded by the RTEMS Project. We have to keep this in mind when we introduce a requirements management work flow which should be followed by RTEMS community members and new contributors.

We have to put in some effort first into the reconstruction of software requirements through reverse engineering using the RTEMS documentation, test cases, sources, standard references, mailing list archives, etc. as input. Writing a technical specification for the complete RTEMS code base is probably a job of several person-years. We have to get started with a moderate feature set (e.g. subset of the Classic API) and extend it based on user demands step by step.

The development of the technical specification will take place in two phases. The first phase tries to establish an initial technical specification for an initial feature set. This technical specification will be integrated into RTEMS as a big chunk. In the second phase the technical specification is modified through arranged procedures. There will be procedures

- to modify existing requirements,
- add new requirements, and
- mark requirements as obsolete.

All procedures should be based on a peer review principles.

## 5.1 Requirements for Requirements

### 5.1.1 Identification

Each requirement shall have a unique identifier (UID). The question is in which scope should it be unique? Ideally, it should be universally unique. Therefore all UIDs used to link one specification item to another should use relative UIDs. This ensures that the RTEMS requirements can be referenced easily in larger systems though a system-specific prefix. The standard ECSS-E-ST-10-06C recommends in section 8.2.6 that the identifier should reflect the type of the requirement and the life profile situation. Other standards may have other recommendations. To avoid a bias of RTEMS in the direction of ECSS, this recommendation will not be followed.

The *absolute UID* of a specification item (for example a requirement) is defined by a leading `/` and the path of directories from the specification base directory to the file of the item separated by `/` characters and the file name without the `.yml` extension. For example, a specification item contained in the file `build/cpukit/librtemscpu.yml` inside a `spec` directory has the absolute UID of `/build/cpukit/librtemscpu`.

The *relative UID* to a specification item is defined by the path of directories from the file containing the source specification item to the file of the destination item separated by `/` characters and the file name of the destination item without the `.yml` extension. For example the relative UID from `/build/bsps/sparc/leon3/grp` to `/build/bsps/bspopts` is `../../bspopts`.

Basically, the valid characters of an UID are determined by the file system storing the item files. By convention, UID characters shall be restricted to the following set defined by the regular expression `[a-zA-Z0-9_-]+`. Use `-` as a separator inside an UID part.

In documents the URL-like prefix `spec:` shall be used to indicated specification item UIDs.

The UID scheme for RTEMS requirements shall be component based. For example, the UID `spec:/classic/task/create-err-invaddr` may specify that the `rtems_task_create()` directive shall return a status of `RTEMS_INVALID_ADDRESS` if the `id` parameter is `NULL`.

A initial requirement item hierarchy could be this:

- build (building RTEMS BSPs and libraries)
- acfg (application configuration groups)
  - opt (application configuration options)
- classic
  - task
    - create-* (requirements for `rtems_task_create()`)
    - delete-* (requirements for `rtems_task_delete()`)
    - exit-* (requirements for `rtems_task_exit()`)
    - getaff-* (requirements for `rtems_task_get_affinity()`)
    - getpri-* (requirements for `rtems_task_get_priority()`)
    - getsched-* (requirements for `rtems_task_get_scheduler()`)
    - ident-* (requirements for `rtems_task_ident()`)
    - issusp-* (requirements for `rtems_task_is_suspended()`)
    - iter-* (requirements for `rtems_task_iterate()`)

        - mode-* (requirements for `rtems_task_mode()`)
        - restart-* (requirements for `rtems_task_restart()`)
        - resume* (requirements for `rtems_task_resume()`)
        - self* (requirements for `rtems_task_self()`)
        - setaff-* (requirements for `rtems_task_set_affinity()`)
        - setpri-* (requirements for `rtems_task_set_priority()`)
        - setsched* (requirements for `rtems_task_set_scheduler()`)
        - start-* (requirements for `rtems_task_start()`)
        - susp-* (requirements for `rtems_task_suspend()`)
        - wkafter-* (requirements for `rtems_task_wake_after()`)
        - wkwhen-* (requirements for `rtems_task_wake_when()`)
    - sema
        - ...
- posix
- ...

A more detailed naming scheme and guidelines should be established. We have to find the right balance between the length of UIDs and self-descriptive UIDs. A clear scheme for all Classic API managers may help to keep the UIDs short and descriptive.

The specification of the validation of requirements should be maintained also by specification items. For each requirement directory there should be a validation subdirectory named *test*, e.g. `spec/classic/task/test`. A test specification directory may contain also validations by analysis, by inspection, and by design, see *Requirement Validation* (page 23).

### 5.1.2 Level of Requirements

The level of a requirement shall be expressed with one of the verbal forms listed below and nothing else. The level of requirements are derived from RFC 2119 [Bra97] and ECSS-E-ST-10-06C [ECS09].

#### 5.1.2.1 Absolute Requirements

Absolute requirements shall be expressed with the verbal form *shall* and no other terms.

#### 5.1.2.2 Absolute Prohibitions

Absolute prohibitions shall be expressed with the verbal form *shall not* and no other terms.

> **Warning**
>
> Absolute prohibitions may be difficult to validate. They should not be used.

#### 5.1.2.3 Recommendations

Recommendations shall be expressed with the verbal forms *should* and *should not* and no other terms with guidance from RFC 2119:

> SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
>
> SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

#### 5.1.2.4 Permissions

Permissions shall be expressed with the verbal form *may* and no other terms with guidance from RFC 2119:

> MAY This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

#### 5.1.2.5 Possibilities and Capabilities

Possibilities and capabilities shall be expressed with the verbal form *can* and no other terms.

### 5.1.3 Syntax

Use the Easy Approach to Requirements Syntax (*EARS*) to formulate requirements. A recommended reading list to get familiar with this approach is [MWHN09], [MW10], [MWGU16], and Alisair Mavin's web site. The patterns are:

- Ubiquitous

  The <system name> shall <system response>.

- Event-driven

  **When** <trigger>, the <system name> shall <system response>.

- State-driven

  **While** <pre-condition>, the <system name> shall <system response>.

- Unwanted behaviour

  **If** <trigger>, **then** the <system name> shall <system response>.

- Optional

  **Where** <feature is included>, the <system name> shall <system response>.

- Complex

  **Where** <feature 0 is included>, **where** <feature 1 is included>, ..., **where** <feature *n* is included>, **while** <pre-condition 0>, **while** <pre-condition 1>, ..., **while** <pre-condition *m*>, **when** <trigger>, the <system name> shall <system response>.

  **Where** <feature 0 is included>, **where** <feature 1 is included>, ..., **where** <feature *n* is included>, **while** <pre-condition 0>, **while** <pre-condition 1>, ..., **while** <pre-condition *m*>, **if** <trigger>, **then** the <system name> shall <system response>.

The optional pattern should be only used for application configuration options. The goal is to use the `enabled-by` attribute to enable or disable requirements based on configuration parameters that define the RTEMS artefacts used to build an application executable (header files, libraries, linker command files). Such configuration parameters are for example the architecture, the platform, CPU port options, and build configuration options (e.g. uniprocessor vs. SMP).

### 5.1.4 Wording Restrictions

To prevent the expression of imprecise requirements, the following terms shall not be used in requirement formulations:

- “acceptable”
- “adequate”
- “almost always”
- “and/or”
- “appropriate”
- “approximately”
- “as far as possible”
- “as much as practicable”
- “best”
- “best possible”
- “easy”
- “efficient”
- “e.g.”
- “enable”
- “enough”
- “etc.”
- “few”
- “first rate”
- “flexible”
- “generally”

- “goal”
- “graceful”
- “great”
- “greatest”
- “ideally”
- “i.e.”
- “if possible”
- “in most cases”
- “large”
- “many”
- “maximize”
- “minimize”
- “most”
- “multiple”
- “necessary”
- “numerous”
- “optimize”
- “ought to”
- “probably”
- “quick”
- “rapid”
- “reasonably”
- “relevant”
- “robust”
- “satisfactory”
- “several”
- “shall be included but not limited to”
- “simple”
- “small”
- “some”
- “state-of-the-art”.
- “sufficient”
- “suitable”
- “support”

- “systematically”
- “transparent”
- “typical”
- “user-friendly”
- “usually”
- “versatile”
- “when necessary”

For guidelines to avoid these terms see Table 11-2, “Some ambiguous terms to avoid in requirements” in [WB13]. There should be some means to enforce that these terms are not used, e.g. through a client-side pre-commit Git hook, a server-side pre-receive Git hook, or some scripts run by special build commands.

### 5.1.5 Separate Requirements

Requirements shall be stated separately. A bad example is:

**spec:/classic/task/create**
The task create directive shall evaluate the parameters, allocate a task object and initialize it.

To make this a better example, it should be split into separate requirements:

**spec:/classic/task/create**
When the task create directive is called with valid parameters and a free task object exists, the task create directive shall assign the identifier of an initialized task object to the `id` parameter and return the `RTEMS_SUCCESSFUL` status.

**spec:/classic/task/create-err-toomany**
If no free task objects exists, the task create directive shall return the `RTEMS_TOO_MANY` status.

**spec:/classic/task/create-err-invaddr**
If the `id` parameter is `NULL`, the task create directive shall return the `RTEMS_INVALID_ADDRESS` status.

**spec:/classic/task/create-err-invname**
If the `name` parameter is invalid, the task create directive shall return the `RTEMS_INVALID_NAME` status.

. . .

### 5.1.6 Conflict Free Requirements

Requirements shall not be in conflict with each other inside a specification. A bad example is:

**spec:/classic/sema/mtx-obtain-wait**
When a mutex is not available, the mutex obtain directive shall enqueue the calling thread on the wait queue of the mutex.

**spec:/classic/sema/mtx-obtain-err-unsat**
If a mutex is not available, the mutex obtain directive shall return the RTEMS_UNSATISFIED status.

To resolve this conflict, a condition may be added:

**spec:/classic/sema/mtx-obtain-wait**
When a mutex is not available and the RTEMS_WAIT option is set, the mutex obtain directive shall enqueue the calling thread on the wait queue of the mutex.

**spec:/classic/sema/mtx-obtain-err-unsat**
If a mutex is not available, when the RTEMS_WAIT option is not set, the mutex obtain directive shall return the RTEMS_UNSATISFIED status.

### 5.1.7 Use of Project-Specific Terms and Abbreviations

All project-specific terms and abbreviations used to formulate requirements shall be defined in the project glossary.

### 5.1.8 Justification of Requirements

Each requirement shall have a rationale or justification recorded in a dedicated section of the requirement file. See `rationale` attribute for *Specification Items* (page 24).

### 5.1.9 Requirement Validation

The validation of each *Requirement Item Type* (page 48) item shall be accomplished by one or more specification items of the types *Test Case Item Type* (page 60) or *Requirement Validation Item Type* (page 57) through a link from the validation item to the requirement item with the *Requirement Validation Link Role* (page 97).

Validation by test is strongly recommended. The choice of any other validation method shall be strongly justified. The requirements author is obligated to provide the means to validate the requirement with detailed instructions.

### 5.1.10 Resources and Performance

Normally, resource and performance requirements are formulated like this:

- The resource U shall need less than V storage units.
- The operation Y shall complete within X time units.

Such statements are difficult to make for a software product like RTEMS which runs on many different target platforms in various configurations. So, the performance requirements of RTEMS shall be stated in terms of benchmarks. The benchmarks are run on the project-specific target platform and configuration. The results obtained by the benchmark runs are reported in a human readable presentation. The application designer can then use the benchmark results to determine if its system performance requirements are met. The benchmarks shall be executed under different environment conditions, e.g. varying cache states (dirty, empty, valid) and system bus load generated by other processors. The application designer shall have the ability to add additional environment conditions, e.g. system bus load by DMA engines or different system bus arbitration schemes.

To catch resource and performance regressions via test suite runs there shall be a means to specify threshold values for the measured quantities. The threshold values should be provided for each validation platform. How this can be done and if the threshold values are maintained by the RTEMS Project is subject to discussion.

## 5.2 Specification Items

### 5.2.1 Specification Item Hierarchy

The specification item types have the following hierarchy:

- *Root Item Type* (page 25)
  - *Build Item Type* (page 26)
    - *Build Ada Test Program Item Type* (page 27)
    - *Build BSP Item Type* (page 28)
    - *Build Configuration File Item Type* (page 30)
    - *Build Configuration Header Item Type* (page 30)
    - *Build Group Item Type* (page 31)
    - *Build Library Item Type* (page 32)
    - *Build Objects Item Type* (page 33)
    - *Build Option Item Type* (page 34)
    - *Build Script Item Type* (page 35)
    - *Build Start File Item Type* (page 37)
    - *Build Test Program Item Type* (page 38)
  - *Constraint Item Type* (page 39)
  - *Glossary Item Type* (page 39)
    - *Glossary Group Item Type* (page 40)
    - *Glossary Term Item Type* (page 40)
  - *Interface Item Type* (page 40)
    - *Application Configuration Group Item Type* (page 41)
    - *Application Configuration Option Item Type* (page 41)
      - *Application Configuration Feature Enable Option Item Type* (page 42)
      - *Application Configuration Feature Option Item Type* (page 42)
      - *Application Configuration Value Option Item Type* (page 42)
    - *Interface Compound Item Type* (page 42)
    - *Interface Define Item Type* (page 43)
    - *Interface Domain Item Type* (page 43)
    - *Interface Enum Item Type* (page 43)
    - *Interface Enumerator Item Type* (page 44)
    - *Interface Forward Declaration Item Type* (page 44)
    - *Interface Function or Macro Item Type* (page 44)
    - *Interface Group Item Type* (page 45)

### 5.2.2 Specification Item Types

#### 5.2.2.1 Root Item Type

The technical specification of RTEMS will contain for example requirements, specializations of requirements, interface specifications, test suites, test cases, and requirement validations. These things will be called *specification items* or just *items* if it is clear from the context.

The specification items are stored in files in *YAML* format with a defined set of key-value pairs called attributes. Each attribute key name shall be a *Name* (page 90). In particular, key names which begin with an underscore (_) are reserved for internal use in tools.

This is the root specification item type. All explicit attributes shall be specified. The explicit attributes for this type are:

**SPDX-License-Identifier**
: The attribute value shall be a *SPDX License Identifier* (page 99). It shall be the license of the item.

**copyrights**
The attribute value shall be a list. Each list element shall be a *Copyright* (page 77). It shall be the list of copyright statements of the item.

**enabled-by**
The attribute value shall be an *Enabled-By Expression* (page 77). It shall define the conditions under which the item is enabled.

**links**
The attribute value shall be a list. Each list element shall be a *Link* (page 89).

**type**
The attribute value shall be a *Name* (page 90). It shall be the item type. The selection of types and the level of detail depends on a particular standard and product model. We need enough flexibility to be in line with ECSS-E-ST-10-06 and possible future applications of other standards. The item type may be refined further with additional type-specific subtypes.

This type is refined by the following types:

- *Build Item Type* (page 26)
- *Constraint Item Type* (page 39)
- *Glossary Item Type* (page 39)
- *Interface Item Type* (page 40)
- *Proxy Item Types* (page 48)
- *Requirement Item Type* (page 48)
- *Requirement Validation Item Type* (page 57)
- *Runtime Measurement Test Item Type* (page 58)
- *Specification Item Type* (page 59)
- *Test Case Item Type* (page 60)
- *Test Suite Item Type* (page 61)

#### 5.2.2.2 Build Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `build`. This set of attributes specifies a build item. Only the `build-type` attribute is mandatory. The explicit attributes for this type are:

**build-type**
The attribute value shall be a *Name* (page 90). It shall be the build item type.

**extra-files**
The attribute value shall be a list of strings. If the value is present, it shall be the list of extra files associated with the item.

This type is refined by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build BSP Item Type* (page 28)
- *Build Configuration File Item Type* (page 30)
- *Build Configuration Header Item Type* (page 30)

- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Option Item Type* (page 34)
- *Build Script Item Type* (page 35)
- *Build Start File Item Type* (page 37)
- *Build Test Program Item Type* (page 38)

#### 5.2.2.3 Build Ada Test Program Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `ada-test-program`. This set of attributes specifies an Ada test program executable to build. Test programs may use additional objects provided by *Build Objects Item Type* (page 33) items. Test programs have an implicit `enabled-by` attribute value which is controlled by the `set-test-state` *Build Option Action* (page 71) of an *Build Option Item Type* (page 34) item. If the test state is set to `exclude`, then the test program is not built. All explicit attributes shall be specified. The explicit attributes for this type are:

**ada-main**
The attribute value shall be a string. It shall be the path to the Ada main body file.

**ada-object-directory**
The attribute value shall be a string. It shall be the path to the Ada object directory (`-D` option value for `gnatmake`).

**adaflags**
The attribute value shall be a list of strings. It shall be a list of options for the Ada compiler.

**adaincludes**
The attribute value shall be a list of strings. It shall be a list of Ada include paths.

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**ldflags**
The attribute value shall be a list. Each list element shall be a *Build Linker Option* (page 71).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

**stlib**
The attribute value shall be a list. Each list element shall be a *Build Link Static Library Directive* (page 70).

**target**
The attribute value shall be a *Build Target* (page 76).

**use-after**
The attribute value shall be a list. Each list element shall be a *Build Use After Directive* (page 77).

**use-before**
The attribute value shall be a list. Each list element shall be a *Build Use Before Directive* (page 77).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
ada-main: testsuites/ada/samples/hello/hello.adb
ada-object-directory: testsuites/ada/samples/hello
adaflags: []
adaincludes:
- cpukit/include/adainclude
- testsuites/ada/support
build-type: ada-test-program
cflags: []
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
cxxflags: []
enabled-by: true
includes: []
ldflags: []
links: []
source:
- testsuites/ada/samples/hello/init.c
stlib: []
target: testsuites/ada/ada_hello.exe
type: build
use-after: []
use-before: []
```

#### 5.2.2.4 Build BSP Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `bsp`. This set of attributes specifies a base BSP variant to build. All explicit attributes shall be specified. The explicit attributes for this type are:

**arch**
The attribute value shall be a string. It shall be the target architecture of the BSP.

**bsp**
The attribute value shall be a string. It shall be the base BSP variant name.

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**family**
The attribute value shall be a string. It shall be the BSP family name. The name shall be the last directory of the path to the BSP sources.

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**install**
The attribute value shall be a list. Each list element shall be a *Build Install Directive* (page 70).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
arch: myarch
bsp: mybsp
build-type: bsp
cflags: []
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
enabled-by: true
family: mybsp
includes: []
install:
- destination: ${BSP_INCLUDEDIR}
  source:
  - bsps/myarch/mybsp/include/bsp.h
  - bsps/myarch/mybsp/include/tm27.h
- destination: ${BSP_INCLUDEDIR}/bsp
  source:
  - bsps/myarch/mybsp/include/bsp/irq.h
- destination: ${BSP_LIBDIR}
  source:
  - bsps/myarch/mybsp/start/linkcmds
links:
- role: build-dependency
  uid: ../../obj
- role: build-dependency
  uid: ../../opto2
- role: build-dependency
  uid: abi
- role: build-dependency
  uid: obj
```

(continues on next page)

(continued from previous page)

```
- role: build-dependency
  uid: ../start
- role: build-dependency
  uid: ../../bspopts
source:
- bsps/myarch/mybsp/start/bspstart.c
type: build
```

#### 5.2.2.5 Build Configuration File Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `config-file`. This set of attributes specifies a configuration file placed in the build tree. The configuration file is generated during the configure command execution and is placed in the build tree. All explicit attributes shall be specified. The explicit attributes for this type are:

**content**
The attribute value shall be a string. It shall be the content of the configuration file. A `${VARIABLE}` substitution is performed during the configure command execution using the variables of the configuration set. Use `$$` for a plain `$` character. To have all variables from sibling items available for substitution it is recommended to link them in the proper order.

**install-path**
The attribute value shall be a *Build Install Path* (page 70).

**target**
The attribute value shall be a *Build Target* (page 76).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
build-type: config-file
content: |
  # ...
  Name: ${ARCH}-rtems${__RTEMS_MAJOR__}-${BSP_NAME}
  # ...
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
install-path: ${PREFIX}/lib/pkgconfig
links: []
target: ${ARCH}-rtems${__RTEMS_MAJOR__}-${BSP_NAME}.pc
type: build
```

#### 5.2.2.6 Build Configuration Header Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `config-header`. This set of attributes specifies configuration header file. The configuration header file is generated during configure command execution and is placed in the build tree. All collected configuration defines are written to the configuration header file during the configure command execution. To have all configuration defines from sibling items available it is recommended to link them in the proper order. All explicit attributes shall be specified. The explicit attributes for this type are:

**guard**
The attribute value shall be a string. It shall be the header guard define.

**include-headers**
The attribute value shall be a list of strings. It shall be a list of header files to include via `#include <...>`.

**install-path**
The attribute value shall be a *Build Install Path* (page 70).

**target**
The attribute value shall be a *Build Target* (page 76).

#### 5.2.2.7 Build Group Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `group`. This set of attributes provides a means to aggregate other build items and modify the build item context which is used by referenced build items. The `includes`, `ldflags`, `objects`, and `use` variables of the build item context are updated by the corresponding attributes of the build group. All explicit attributes shall be specified. The explicit attributes for this type are:

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**install**
The attribute value shall be a list. Each list element shall be a *Build Install Directive* (page 70).

**ldflags**
The attribute value shall be a list of strings. It shall be a list of options for the linker. They are used to link executables referenced by this item.

**use-after**
The attribute value shall be a list. Each list element shall be a *Build Use After Directive* (page 77).

**use-before**
The attribute value shall be a list. Each list element shall be a *Build Use Before Directive* (page 77).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
build-type: group
cflags: []
copyrights:
```

(continues on next page)

(continued from previous page)

```
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
cxxflags: []
enabled-by:
- BUILD_TESTS
- BUILD_SAMPLES
includes:
- testsuites/support/include
install: []
ldflags:
- -Wl,--wrap=printf
- -Wl,--wrap=puts
links:
- role: build-dependency
  uid: ticker
type: build
use-after: []
use-before:
- rtemstest
```

#### 5.2.2.8 Build Library Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `library`. This set of attributes specifies a static library. Library items may use additional objects provided by *Build Objects Item Type* (page 33) items through the build dependency links of the item. All explicit attributes shall be specified. The explicit attributes for this type are:

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**install**
The attribute value shall be a list. Each list element shall be a *Build Install Directive* (page 70).

**install-path**
The attribute value shall be a *Build Install Path* (page 70).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

**target**
The attribute value shall be a *Build Target* (page 76). It shall be the name of the static library, e.g. `z` for `libz.a`.

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
build-type: library
cflags:
- -Wno-pointer-sign
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
cxxflags: []
enabled-by: true
includes:
- cpukit/libfs/src/jffs2/include
install:
- destination: ${BSP_INCLUDEDIR}/rtems
  source:
  - cpukit/include/rtems/jffs2.h
install-path: ${BSP_LIBDIR}
links: []
source:
- cpukit/libfs/src/jffs2/src/build.c
target: jffs2
type: build
```

#### 5.2.2.9 Build Objects Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `objects`. This set of attributes specifies a set of object files used to build static libraries or test programs. Objects Items must not be included on multiple paths through the build dependency graph with identical build options. Violating this can cause race conditions in the build system due to duplicate installs and multiple instances of build tasks. All explicit attributes shall be specified. The explicit attributes for this type are:

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**install**
The attribute value shall be a list. Each list element shall be a *Build Install Directive* (page 70).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
build-type: objects
cflags: []
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
cxxflags: []
enabled-by: true
includes: []
install:
- destination: ${BSP_INCLUDEDIR}/bsp
  source:
  - bsps/include/bsp/bootcard.h
  - bsps/include/bsp/default-initial-extension.h
  - bsps/include/bsp/fatal.h
links: []
source:
- bsps/shared/start/bootcard.c
- bsps/shared/rtems-version.c
type: build
```

#### 5.2.2.10 Build Option Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `option`. This set of attributes specifies a build option. The following explicit attributes are mandatory:

- `actions`
- `default`
- `description`

The explicit attributes for this type are:

**actions**
The attribute value shall be a list. Each list element shall be a *Build Option Action* (page 71). Each action operates on the *action value* handed over by a previous action and action-specific attribute values. The actions pass the processed action value to the next action in the list. The first action starts with an action value of `None`. The actions are carried out during the configure command execution.

**default**
The attribute value shall be a list. Each list element shall be a *Build Option Value* (page 75). It shall be the list of default values of the option. When a default value is needed, the first value on the list which is enabled according to the enabled set is chosen. If no value is enabled, then the default value is `null`.

**description**
The attribute value shall be an optional string. It shall be the description of the option.

**format**
The attribute value shall be an optional string. It shall be a Python format string, for example `'{}'` or `'{:#010x}'`.

**name**
The attribute value shall be a *Build Option Name* (page 75).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
actions:
- get-integer: null
- define: null
build-type: option
copyrights:
- Copyright (C) 2020, 2022 embedded brains GmbH & Co. KG
default:
- enabled-by:
  - bsps/powerpc/motorola_powerpc
  - m68k/m5484FireEngine
  - powerpc/hsc_cm01
  value: 9600
- enabled-by: m68k/COBRA5475
  value: 19200
- enabled-by: true
  value: 115200
description: |
  Default baud for console and other serial devices.
enabled-by: true
format: '{}'
links: []
name: BSP_CONSOLE_BAUD
type: build
```

#### 5.2.2.11 Build Script Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `script`. This set of attributes specifies a build script. The optional attributes may be required by commands executed through the scripts. The following explicit attributes are mandatory:

- `do-build`
- `do-configure`
- `prepare-build`
- `prepare-configure`

The explicit attributes for this type are:

**asflags**
The attribute value shall be a list. Each list element shall be a *Build Assembler Option* (page 68).

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option*

(page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**do-build**
The attribute value shall be an optional string. If this script shall execute, then it shall be Python code which is executed via `exec()` in the context of the `do_build()` method of the `wscript`. A local variable `bld` is available with the `waf` build context. A local variable `bic` is available with the build item context.

**do-configure**
The attribute value shall be an optional string. If this script shall execute, then it shall be Python code which is executed via `exec()` in the context of the `do_configure()` method of the `wscript`. A local variable `conf` is available with the `waf` configuration context. A local variable `cic` is available with the configuration item context.

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**ldflags**
The attribute value shall be a list. Each list element shall be a *Build Linker Option* (page 71).

**prepare-build**
The attribute value shall be an optional string. If this script shall execute, then it shall be Python code which is executed via `exec()` in the context of the `prepare_build()` method of the `wscript`. A local variable `bld` is available with the `waf` build context. A local variable `bic` is available with the build item context.

**prepare-configure**
The attribute value shall be an optional string. If this script shall execute, then it shall be Python code which is executed via `exec()` in the context of the `prepare_configure()` method of the `wscript`. A local variable `conf` is available with the `waf` configuration context. A local variable `cic` is available with the configuration item context.

**stlib**
The attribute value shall be a list. Each list element shall be a *Build Link Static Library Directive* (page 70).

**target**
The attribute value shall be a *Build Target* (page 76).

**use-after**
The attribute value shall be a list. Each list element shall be a *Build Use After Directive* (page 77).

**use-before**
The attribute value shall be a list. Each list element shall be a *Build Use Before Directive* (page 77).

Please have a look at the following example:

```
1 SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
2 build-type: script
3 copyrights:
4 - Copyright (C) 2020 embedded brains GmbH & Co. KG
```

(continues on next page)

(continued from previous page)

```
default: null
default-by-variant: []
do-build: |
  bld.install_as(
      "${BSP_LIBDIR}/linkcmds",
      "bsps/" + bld.env.ARCH + "/" + bld.env.BSP_FAMILY +
      "/start/linkcmds." + bld.env.BSP_BASE
  )
do-configure: |
  conf.env.append_value(
      "LINKFLAGS",
      ["-qnolinkcmds", "-T", "linkcmds." + conf.env.BSP_BASE]
  )
enabled-by: true
links: []
prepare-build: null
prepare-configure: null
type: build
```

#### 5.2.2.12 Build Start File Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `start-file`. This set of attributes specifies a start file to build. A start file is used to link an executable. All explicit attributes shall be specified. The explicit attributes for this type are:

**asflags**
The attribute value shall be a list. Each list element shall be a *Build Assembler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**install-path**
The attribute value shall be a *Build Install Path* (page 70).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

**target**
The attribute value shall be a *Build Target* (page 76).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
asflags: []
build-type: start-file
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
```

(continues on next page)

(continued from previous page)

```
enabled-by: true
includes: []
install-path: ${BSP_LIBDIR}
links: []
source:
- bsps/sparc/shared/start/start.S
target: start.o
type: build
```

#### 5.2.2.13 Build Test Program Item Type

This type refines the *Build Item Type* (page 26) through the `build-type` attribute if the value is `test-program`. This set of attributes specifies a test program executable to build. Test programs may use additional objects provided by *Build Objects Item Type* (page 33) items. Test programs have an implicit `enabled-by` attribute value which is controlled by the `set-test-state` *Build Option Action* (page 71) of an *Build Option Item Type* (page 34) item. If the test state is set to `exclude`, then the test program is not built. All explicit attributes shall be specified. The explicit attributes for this type are:

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**cppflags**
The attribute value shall be a list. Each list element shall be a *Build C Preprocessor Option* (page 68).

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**features**
The attribute value shall be a string. It shall be the `waf` build features for this test program.

**includes**
The attribute value shall be a list. Each list element shall be a *Build Include Path* (page 69).

**ldflags**
The attribute value shall be a list. Each list element shall be a *Build Linker Option* (page 71).

**source**
The attribute value shall be a list. Each list element shall be a *Build Source* (page 76).

**stlib**
The attribute value shall be a list. Each list element shall be a *Build Link Static Library Directive* (page 70).

**target**
The attribute value shall be a *Build Target* (page 76).

**use-after**
The attribute value shall be a list. Each list element shall be a *Build Use After Directive* (page 77).

**use-before**
The attribute value shall be a list. Each list element shall be a *Build Use Before Directive* (page 77).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
build-type: test-program
cflags: []
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
cppflags: []
cxxflags: []
enabled-by: true
features: c cprogram
includes: []
ldflags: []
links: []
source:
- testsuites/samples/ticker/init.c
- testsuites/samples/ticker/tasks.c
stlib: []
target: testsuites/samples/ticker.exe
type: build
use-after: []
use-before: []
```

#### 5.2.2.14 Constraint Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `constraint`. This set of attributes specifies a constraint. All explicit attributes shall be specified. The explicit attributes for this type are:

**rationale**
The attribute value shall be an optional string. If the value is present, then it shall state the rationale or justification of the constraint.

**text**
The attribute value shall be a *Requirement Text* (page 95). It shall state the constraint.

#### 5.2.2.15 Glossary Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `glossary`. This set of attributes specifies a glossary item. All explicit attributes shall be specified. The explicit attributes for this type are:

**glossary-type**
The attribute value shall be a *Name* (page 90). It shall be the glossary item type.

This type is refined by the following types:

- *Glossary Group Item Type* (page 40)
- *Glossary Term Item Type* (page 40)

#### 5.2.2.16 Glossary Group Item Type

This type refines the *Glossary Item Type* (page 39) through the `glossary-type` attribute if the value is `group`. This set of attributes specifies a glossary group. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be the human readable name of the glossary group.

**text**
The attribute value shall be a string. It shall state the requirement for the glossary group.

#### 5.2.2.17 Glossary Term Item Type

This type refines the *Glossary Item Type* (page 39) through the `glossary-type` attribute if the value is `term`. This set of attributes specifies a glossary term. All explicit attributes shall be specified. The explicit attributes for this type are:

**term**
The attribute value shall be a string. It shall be the glossary term.

**text**
The attribute value shall be a string. It shall be the definition of the glossary term.

#### 5.2.2.18 Interface Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `interface`. This set of attributes specifies an interface specification item. Interface items shall specify the interface of the software product to other software products and the hardware. Use *Interface Domain Item Type* (page 43) items to specify interface domains, for example the *API*, C language, compiler, interfaces to the implementation, and the hardware. All explicit attributes shall be specified. The explicit attributes for this type are:

**index-entries**
The attribute value shall be a list of strings. It shall be a list of additional document index entries. A document index entry derived from the interface name is added automatically.

**interface-type**
The attribute value shall be a *Name* (page 90). It shall be the interface item type.

This type is refined by the following types:

- *Application Configuration Group Item Type* (page 41)
- *Application Configuration Option Item Type* (page 41)
- *Interface Compound Item Type* (page 42)
- *Interface Define Item Type* (page 43)
- *Interface Domain Item Type* (page 43)
- *Interface Enum Item Type* (page 43)
- *Interface Enumerator Item Type* (page 44)
- *Interface Forward Declaration Item Type* (page 44)
- *Interface Function or Macro Item Type* (page 44)

- *Interface Group Item Type* (page 45)
- *Interface Header File Item Type* (page 45)
- *Interface Typedef Item Type* (page 45)
- *Interface Unspecified Header File Item Type* (page 46)
- *Interface Unspecified Item Type* (page 46)
- *Interface Variable Item Type* (page 47)
- *Register Block Item Type* (page 47)

#### 5.2.2.19 Application Configuration Group Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `appl-config-group`. This set of attributes specifies an application configuration group. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be a string. It shall be the description of the application configuration group.

**name**
The attribute value shall be a string. It shall be human readable name of the application configuration group.

**text**
The attribute value shall be a *Requirement Text* (page 95). It shall state the requirement for the application configuration group.

#### 5.2.2.20 Application Configuration Option Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `appl-config-option`. This set of attributes specifies an application configuration option. All explicit attributes shall be specified. The explicit attributes for this type are:

**appl-config-option-type**
The attribute value shall be a *Name* (page 90). It shall be the application configuration option type.

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be an *Application Configuration Option Name* (page 67).

**notes**
The attribute value shall be an *Interface Notes* (page 87).

This type is refined by the following types:

- *Application Configuration Feature Enable Option Item Type* (page 42)
- *Application Configuration Feature Option Item Type* (page 42)
- *Application Configuration Value Option Item Type* (page 42)

#### 5.2.2.21 Application Configuration Feature Enable Option Item Type

This type refines the *Application Configuration Option Item Type* (page 41) through the `appl-config-option-type` attribute if the value is `feature-enable`. This set of attributes specifies an application configuration feature enable option.

#### 5.2.2.22 Application Configuration Feature Option Item Type

This type refines the *Application Configuration Option Item Type* (page 41) through the `appl-config-option-type` attribute if the value is `feature`. This set of attributes specifies an application configuration feature option. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be a string. It shall describe what happens if the configuration option is undefined.

#### 5.2.2.23 Application Configuration Value Option Item Type

This type refines the following types:

- *Application Configuration Option Item Type* (page 41) through the `appl-config-option-type` attribute if the value is `initializer`
- *Application Configuration Option Item Type* (page 41) through the `appl-config-option-type` attribute if the value is `integer`

This set of attributes specifies application configuration initializer or integer option. All explicit attributes shall be specified. The explicit attributes for this type are:

**default-value**
The attribute value shall be an *Integer or String* (page 80). It shall describe the default value of the application configuration option.

#### 5.2.2.24 Interface Compound Item Type

This type refines the following types:

- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `struct`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `union`

This set of attributes specifies a compound (struct or union). All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be a list. Each list element shall be an *Interface Compound Member Definition Directive* (page 82).

**definition-kind**
The attribute value shall be an *Interface Compound Definition Kind* (page 81).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the compound (struct or union).

**notes**
The attribute value shall be an *Interface Notes* (page 87).

#### 5.2.2.25 Interface Define Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `define`. This set of attributes specifies a define. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be an *Interface Definition Directive* (page 83).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the define.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

#### 5.2.2.26 Interface Domain Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `domain`. This set of attributes specifies an interface domain. Interface items are placed into domains through links with the *Interface Placement Link Role* (page 88). All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be a string. It shall be the description of the domain

**name**
The attribute value shall be a string. It shall be the human readable name of the domain.

#### 5.2.2.27 Interface Enum Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `enum`. This set of attributes specifies an enum. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition-kind**
The attribute value shall be an *Interface Enum Definition Kind* (page 85).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the enum.

**notes**
The attribute value shall be an *Interface Description* (page 83).

#### 5.2.2.28 Interface Enumerator Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `enumerator`. This set of attributes specifies an enumerator. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be an *Interface Definition Directive* (page 83).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the enumerator.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

#### 5.2.2.29 Interface Forward Declaration Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `forward-declaration`. Items of this type specify a forward declaration. The item shall have exactly one link with the *Interface Target Link Role* (page 89) to an *Interface Compound Item Type* (page 42) item. This link defines the type declared by the forward declaration.

#### 5.2.2.30 Interface Function or Macro Item Type

This type refines the following types:

- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `function`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `macro`

This set of attributes specifies a function or a macro. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be an *Interface Function or Macro Definition Directive* (page 86).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the function or macro.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

**params**
The attribute value shall be a list. Each list element shall be an *Interface Parameter* (page 88).

**return**
The attribute value shall be an *Interface Return Directive* (page 88).

#### 5.2.2.31 Interface Group Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `group`. This set of attributes specifies an interface group. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**description**
The attribute value shall be an *Interface Description* (page 83).

**identifier**
The attribute value shall be an *Interface Group Identifier* (page 87).

**name**
The attribute value shall be a string. It shall be the human readable name of the interface group.

**text**
The attribute value shall be a *Requirement Text* (page 95). It shall state the requirement for the interface group.

#### 5.2.2.32 Interface Header File Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `header-file`. This set of attributes specifies a header file. The item shall have exactly one link with the *Interface Placement Link Role* (page 88) to an *Interface Domain Item Type* (page 43) item. This link defines the interface domain of the header file. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**path**
The attribute value shall be a string. It shall be the path used to include the header file. For example `rtems/confdefs.h`.

**prefix**
The attribute value shall be a string. It shall be the prefix directory path to the header file in the interface domain. For example `cpukit/include`.

#### 5.2.2.33 Interface Typedef Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `typedef`. This set of attributes specifies a typedef. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be an *Interface Definition Directive* (page 83).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the typedef.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

**params**
The attribute value shall be a list. Each list element shall be an *Interface Parameter* (page 88).

**return**
The attribute value shall be an *Interface Return Directive* (page 88).

#### 5.2.2.34 Interface Unspecified Header File Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-header-file`. This set of attributes specifies an unspecified header file. All explicit attributes shall be specified. The explicit attributes for this type are:

**path**
The attribute value shall be a string. It shall be the path used to include the header file. For example `rtems/confdefs.h`.

**references**
The attribute value shall be a list. Each list element shall be an *External Reference* (page 79).

#### 5.2.2.35 Interface Unspecified Item Type

This type refines the following types:

- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-define`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-enum`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-enumerator`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-function`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-group`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-macro`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-object`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-struct`
- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-typedef`

- *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `unspecified-union`

This set of attributes specifies an unspecified interface. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be the name of the unspecified interface.

**references**
The attribute value shall be a list. Each list element shall be an *External Reference* (page 79).

#### 5.2.2.36 Interface Variable Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `variable`. This set of attributes specifies a variable. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be an *Interface Definition Directive* (page 83).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name of the variable.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

#### 5.2.2.37 Register Block Item Type

This type refines the *Interface Item Type* (page 40) through the `interface-type` attribute if the value is `register-block`. This set of attributes specifies a register block. A register block may be used to specify the interface of devices. Register blocks consist of register block members specified by the `definition` attribute. Register block members are either instances of registers specified by the `registers` attribute or instances of other register blocks specified by links with the *Register Block Include Role* (page 93). Registers consists of bit fields (see *Register Bits Definition* (page 92). The register block members are placed into the address space of the device relative to the base address of the register block. Register member offsets and the register block size are specified in units of the address space granule. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**definition**
The attribute value shall be a list. Each list element shall be a *Register Block Member Definition Directive* (page 93).

**description**
The attribute value shall be an *Interface Description* (page 83).

**identifier**
The attribute value shall be an *Interface Group Identifier* (page 87).

**name**
The attribute value shall be a string. It shall be the name of the register block.

**notes**
The attribute value shall be an *Interface Notes* (page 87).

**register-block-group**
The attribute value shall be a string. It shall be the name of the interface group defined for the register block. For the group identifier see the `identifier` attribute.

**register-block-size**
The attribute value shall be an *Optional Integer* (page 91). If the value is present, then it shall be the size of the register block in units of the address space granule.

**register-prefix**
The attribute value shall be an optional string. If the value is present, then it will be used to prefix register bit field names, otherwise the value of the `name` attribute will be used.

**registers**
The attribute value shall be a list. Each list element shall be a *Register Definition* (page 94).

#### 5.2.2.38 Proxy Item Types

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `proxy`. Items of similar characteristics may link to a proxy item through links with the *Proxy Member Link Role* (page 92). A proxy item resolves to the first member item which is enabled. Proxies may be used to provide an interface with a common name and implementations which depend on configuration options. For example, in one configuration a constant could be a compile time constant and in another configuration it could be a read-only object.

#### 5.2.2.39 Requirement Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `requirement`. This set of attributes specifies a requirement. All explicit attributes shall be specified. The explicit attributes for this type are:

**rationale**
The attribute value shall be an optional string. If the value is present, then it shall state the rationale or justification of the requirement.

**references**
The attribute value shall be a list. Each list element shall be an *External Reference* (page 79).

**requirement-type**
The attribute value shall be a *Name* (page 90). It shall be the requirement item type.

**text**
The attribute value shall be a *Requirement Text* (page 95). It shall state the requirement.

This type is refined by the following types:

- *Functional Requirement Item Type* (page 49)
- *Non-Functional Requirement Item Type* (page 54)

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
functional-type: capability
links: []
rationale: |
  It keeps you busy.
requirement-type: functional
text: |
  The system shall do crazy things.
type: requirement
```

#### 5.2.2.40 Functional Requirement Item Type

This type refines the *Requirement Item Type* (page 48) through the `requirement-type` attribute if the value is `functional`. This set of attributes specifies a functional requirement. All explicit attributes shall be specified. The explicit attributes for this type are:

**functional-type**
The attribute value shall be a *Name* (page 90). It shall be the functional type of the requirement.

This type is refined by the following types:

- *Action Requirement Item Type* (page 49)
- *Generic Functional Requirement Item Type* (page 53)

#### 5.2.2.41 Action Requirement Item Type

This type refines the *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `action`. This set of attributes specifies functional requirements and corresponding validation test code. The functional requirements of an action are specified. An action performs a step in a finite state machine. An action is implemented through a function or a macro. The action is performed through a call of the function or an execution of the code of a macro expansion by an actor. The actor is for example a task or an interrupt service routine.

For action requirements which specify the function of an interface, there shall be exactly one link with the *Interface Function Link Role* (page 85) to the interface of the action.

The action requirements are specified by

- a list of pre-conditions, each with a set of states,
- a list of post-conditions, each with a set of states,
- the transition of pre-condition states to post-condition states through the action.

Along with the requirements, the test code to generate a validation test is specified. For an action requirement it is verified that all variations of pre-condition states have a set of post-condition states specified in the transition map. All transitions are covered by the generated test code. All explicit attributes shall be specified. The explicit attributes for this type are:

**post-conditions**
The attribute value shall be a list. Each list element shall be an *Action Requirement Condition*

(page 63).

**pre-conditions**
The attribute value shall be a list. Each list element shall be an *Action Requirement Condition* (page 63).

**skip-reasons**
The attribute value shall be an *Action Requirement Skip Reasons* (page 65).

**test-action**
The attribute value shall be a string. It shall be the test action code.

**test-brief**
The attribute value shall be an optional string. If the value is present, then it shall be the test case brief description.

**test-cleanup**
The attribute value shall be an optional string. If the value is present, then it shall be the test cleanup code. The code is placed in the test action loop body after the test post-condition checks.

**test-context**
The attribute value shall be a list. Each list element shall be a *Test Context Member* (page 108).

**test-context-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test context support code. The context support code is placed at file scope before the test context definition.

**test-description**
The attribute value shall be an optional string. If the value is present, then it shall be the test case description.

**test-header**
The attribute value shall be a *Test Header* (page 108).

**test-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include <...>`.

**test-local-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include "..."`.

**test-prepare**
The attribute value shall be an optional string. If the value is present, then it shall be the early test preparation code. The code is placed in the test action loop body before the test pre-condition preparations.

**test-setup**
The attribute value shall be a *Test Support Method* (page 109).

**test-stop**
The attribute value shall be a *Test Support Method* (page 109).

**test-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test case support code. The support code is placed at file scope before the test case code.

**test-target**
The attribute value shall be a string. It shall be the path to the generated test case source file.

**test-teardown**
The attribute value shall be a *Test Support Method* (page 109).

**transition-map**
The attribute value shall be a list. Each list element shall be an *Action Requirement Transition* (page 65).

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
functional-type: action
links: []
post-conditions:
- name: Status
  states:
  - name: Success
    test-code: |
      /* Check that the status is SUCCESS */
    text: |
      The status shall be SUCCESS.
  - name: Error
    test-code: |
      /* Check that the status is ERROR */
    text: |
      The status shall be ERROR.
  test-epilogue: null
  test-prologue: null
- name: Data
  states:
  - name: Unchanged
    test-code: |
      /* Check that the data is unchanged */
    text: |
      The data shall be unchanged by the action.
  - name: Red
    test-code: |
      /* Check that the data is red */
    text: |
      The data shall be red.
  - name: Green
    test-code: |
      /* Check that the data is green */
    text: |
      The data shall be green.
  test-epilogue: null
  test-prologue: null
```

(continues on next page)

(continued from previous page)

```
pre-conditions:
- name: Data
  states:
  - name: NullPtr
    test-code: |
      /* Set data pointer to NULL */
    text: |
      The data pointer shall be NULL.
  - name: Valid
    test-code: |
      /* Set data pointer to reference a valid data buffer */
    text: |
      The data pointer shall reference a valid data buffer.
  test-epilogue: null
  test-prologue: null
- name: Option
  states:
  - name: Red
    test-code: |
      /* Set option to RED */
    text: |
      The option shall be RED.
  - name: Green
    test-code: |
      /* Set option to GREEN */
    text: |
      The option shall be GREEN.
  test-epilogue: null
  test-prologue: null
requirement-type: functional
skip-reasons: {}
test-action: |
  /* Call the function of the action */
test-brief: null
test-cleanup: null
test-context:
- brief: null
  description: null
  member: void *data
- brief: null
  description: null
  member: option_type option
test-context-support: null
test-description: null
test-header: null
test-includes: []
test-local-includes: []
test-prepare: null
test-setup: null
```

(continues on next page)

(continued from previous page)

```
test-stop: null
test-support: null
test-target: tc-red-green-data.c
test-teardown: null
transition-map:
- enabled-by: true
  post-conditions:
    Status: Error
    Data: Unchanged
  pre-conditions:
    Data: NullPtr
    Option: all
- enabled-by: true
  post-conditions:
    Status: Success
    Data: Red
  pre-conditions:
    Data: Valid
    Option: Red
- enabled-by: true
  post-conditions:
    Status: Success
    Data: Green
  pre-conditions:
    Data: Valid
    Option: Green
rationale: null
references: []
text: |
  ${.:/text-template}
type: requirement
```

#### 5.2.2.42 Generic Functional Requirement Item Type

This type refines the following types:

- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `capability`
- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `dependability-function`
- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `function`
- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `interface-define-not-defined`
- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `operational`
- *Functional Requirement Item Type* (page 49) through the `functional-type` attribute if the value is `safety-function`

Items of this type state a functional requirement with the functional type defined by the specification type refinement.

#### 5.2.2.43 Non-Functional Requirement Item Type

This type refines the *Requirement Item Type* (page 48) through the `requirement-type` attribute if the value is `non-functional`. This set of attributes specifies a non-functional requirement. All explicit attributes shall be specified. The explicit attributes for this type are:

**non-functional-type**
The attribute value shall be a *Name* (page 90). It shall be the non-functional type of the requirement.

This type is refined by the following types:

- *Design Group Requirement Item Type* (page 54)
- *Design Target Item Type* (page 54)
- *Generic Non-Functional Requirement Item Type* (page 55)
- *Runtime Measurement Environment Item Type* (page 55)
- *Runtime Performance Requirement Item Type* (page 56)

#### 5.2.2.44 Design Group Requirement Item Type

This type refines the *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `design-group`. This set of attributes specifies a design group requirement. Design group requirements have an explicit reference to the associated Doxygen group specified by the `identifier` attribute. Design group requirements have an implicit validation by inspection method. The qualification toolchain shall perform the inspection and check that the specified Doxygen group exists in the software source code. All explicit attributes shall be specified. The explicit attributes for this type are:

**identifier**
The attribute value shall be a *Requirement Design Group Identifier* (page 95).

#### 5.2.2.45 Design Target Item Type

This type refines the *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `design-target`. This set of attributes specifies a design *target*. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an optional string. If the value is present, then it shall briefly describe the target.

**description**
The attribute value shall be an optional string. If the value is present, then it shall thoroughly describe the target.

**name**
The attribute value shall be a string. It shall be the target name.

#### 5.2.2.46 Generic Non-Functional Requirement Item Type

This type refines the following types:

- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `build-configuration`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `constraint`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `design`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `documentation`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `interface`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `interface-requirement`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `maintainability`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `performance`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `performance-runtime-limits`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `portability`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `quality`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `reliability`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `resource`
- *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `safety`

Items of this type state a non-functional requirement with the non-functional type defined by the specification type refinement.

#### 5.2.2.47 Runtime Measurement Environment Item Type

This type refines the *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `performance-runtime-environment`. This set of attributes specifies a runtime measurement environment. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be the runtime measurement environment name. See also *Runtime Measurement Environment Name* (page 97).

#### 5.2.2.48 Runtime Performance Requirement Item Type

This type refines the *Non-Functional Requirement Item Type* (page 54) through the `non-functional-type` attribute if the value is `performance-runtime`. The item shall have exactly one link with the *Runtime Measurement Request Link Role* (page 98). A requirement text processor shall support a substitution of `${.:/limit-kind}`:

- For a *Runtime Measurement Value Kind* (page 98) of `min-lower-bound` or `min-upper-bound`, the substitution of `${.:/limit-kind}` shall be "`minimum`".
- For a *Runtime Measurement Value Kind* (page 98) of `mean-lower-bound` or `mean-upper-bound`, the substitution of `${.:/limit-kind}` shall be "`mean`".
- For a *Runtime Measurement Value Kind* (page 98) of `max-lower-bound` or `max-upper-bound`, the substitution of `${.:/limit-kind}` shall be "`maximum`".

A requirement text processor shall support a substitution of `${.:/limit-condition}`:

- For a *Runtime Measurement Value Kind* (page 98) of `min-lower-bound`, `mean-lower-bound`, or `max-lower-bound`, the substitution of `${.:/limit-condition}` shall be "`greater than or equal to <value>`" with `<value>` being the value of the corresponding entry in the *Runtime Measurement Value Table* (page 99).
- For a *Runtime Measurement Value Kind* (page 98) of `min-upper-bound`, `mean-upper-bound`, or `max-upper-bound`, the substitution of `${.:/limit-condition}` shall be "`less than or equal to <value>`" with `<value>` being the value of the corresponding entry in the *Runtime Measurement Value Table* (page 99).

A requirement text processor shall support a substitution of `${.:/environment}`. The value of the substitution shall be "`<environment> environment`" with `<environment>` being the environment of the corresponding entry in the *Runtime Measurement Environment Table* (page 98).

This set of attributes specifies a runtime performance requirement. Along with the requirement, the validation test code to execute a measure runtime request is specified. All explicit attributes shall be specified. The explicit attributes for this type are:

**params**
The attribute value shall be a *Runtime Performance Parameter Set* (page 99).

**test-body**
The attribute value shall be a *Test Support Method* (page 109). It shall provide the code of the measure runtime body handler. In contrast to other methods, this method is mandatory.

**test-cleanup**
The attribute value shall be a *Test Support Method* (page 109). It may provide the code to clean up the measure runtime request. This method is called before the cleanup method of the corresponding *Runtime Measurement Test Item Type* (page 58) item and after the request.

**test-prepare**
The attribute value shall be a *Test Support Method* (page 109). It may provide the code to prepare the measure runtime request. This method is called after the prepare method of the corresponding *Runtime Measurement Test Item Type* (page 58) item and before the request.

**test-setup**
The attribute value shall be a *Test Support Method* (page 109). It may provide the code of the measure runtime setup handler.

**test-teardown**
The attribute value shall be a *Test Support Method* (page 109). It may provide the code of the

measure runtime teardown handler.

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
links:
- role: runtime-measurement-request
  uid: ../val/perf
params: {}
rationale: null
references: []
test-body:
  brief: |
    Get a buffer.
  code: |
    ctx->status = rtems_partition_get_buffer( ctx->part_many, &ctx->buffer );
  description: null
test-cleanup: null
test-prepare: null
test-setup: null
test-teardown:
  brief: |
    Return the buffer.
  code: |
    rtems_status_code sc;

    T_quiet_rsc_success( ctx->status );

    sc = rtems_partition_return_buffer( ctx->part_many, ctx->buffer );
    T_quiet_rsc_success( sc );

    return tic == toc;
  description: null
text: |
  When a partition has exactly ${../val/perf:/params/buffer-count} free
  buffers, the ${.:/limit-kind} runtime of exactly
  ${../val/perf:/params/sample-count} successful calls to
  ${../if/get-buffer:/name} in the ${.:/environment} shall be
  ${.:/limit-condition}.
non-functional-type: performance-runtime
requirement-type: non-functional
type: requirement
```

#### 5.2.2.49 Requirement Validation Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `validation`. This set of attributes provides a requirement validation evidence. The item shall have exactly one link to the validated requirement with the *Requirement Validation Link Role* (page 97). All explicit attributes shall be specified. The explicit attributes for this type are:

**method**
The attribute value shall be a *Name* (page 90). It shall specify the requirement validation method (except validation by test). Validation by test is done through *Test Case Item Type* (page 60) items.

**references**
The attribute value shall be a list. Each list element shall be an *External Reference* (page 79).

**text**
The attribute value shall be a string. It shall provide the validation evidence depending on the validation method:

- *By analysis*: A statement shall be provided how the requirement is met, by analysing static properties of the *software product*.
- *By inspection*: A statement shall be provided how the requirement is met, by inspection of the *source code*.
- *By review of design*: A rationale shall be provided to demonstrate how the requirement is satisfied implicitly by the software design.

This type is refined by the following types:

- *Requirement Validation Method* (page 58)

#### 5.2.2.50 Requirement Validation Method

This type refines the following types:

- *Requirement Validation Item Type* (page 57) through the `method` attribute if the value is `by-analysis`
- *Requirement Validation Item Type* (page 57) through the `method` attribute if the value is `by-inspection`
- *Requirement Validation Item Type* (page 57) through the `method` attribute if the value is `by-review-of-design`

#### 5.2.2.51 Runtime Measurement Test Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `runtime-measurement-test`. This set of attributes specifies a runtime measurement test case. All explicit attributes shall be specified. The explicit attributes for this type are:

**params**
The attribute value shall be a *Runtime Measurement Parameter Set* (page 98).

**test-brief**
The attribute value shall be an optional string. If the value is present, then it shall be the test case brief description.

**test-cleanup**
The attribute value shall be a *Test Support Method* (page 109). If the value is present, then it shall be the measure runtime request cleanup method. The method is called after each measure runtime request.

**test-context**
The attribute value shall be a list. Each list element shall be a *Test Context Member* (page 108).

**test-context-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test context support code. The context support code is placed at file scope before the test context definition.

**test-description**
The attribute value shall be an optional string. If the value is present, then it shall be the test case description.

**test-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include <...>`.

**test-local-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include "..."`.

**test-prepare**
The attribute value shall be a *Test Support Method* (page 109). If the value is present, then it shall be the measure runtime request prepare method. The method is called before each measure runtime request.

**test-setup**
The attribute value shall be a *Test Support Method* (page 109). If the value is present, then it shall be the test case setup fixture method.

**test-stop**
The attribute value shall be a *Test Support Method* (page 109). If the value is present, then it shall be the test case stop fixture method.

**test-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test case support code. The support code is placed at file scope before the test case code.

**test-target**
The attribute value shall be a string. It shall be the path to the generated test case source file.

**test-teardown**
The attribute value shall be a *Test Support Method* (page 109). If the value is present, then it shall be the test case teardown fixture method.

#### 5.2.2.52 Specification Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `spec`. This set of attributes specifies specification types. All explicit attributes shall be specified. The explicit attributes for this type are:

**spec-description**
The attribute value shall be an optional string. It shall be the description of the specification type.

**spec-example**
The attribute value shall be an optional string. If the value is present, then it shall be an example of the specification type.

**spec-info**
The attribute value shall be a *Specification Information* (page 102).

**spec-name**
The attribute value shall be an optional string. It shall be the human readable name of the specification type.

**spec-type**
The attribute value shall be a *Name* (page 90). It shall the specification type.

Please have a look at the following example:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
links:
- role: spec-member
  uid: root
- role: spec-refinement
  spec-key: type
  spec-value: example
  uid: root
spec-description: null
spec-example: null
spec-info:
  dict:
    attributes:
      an-example-attribute:
        description: |
          It shall be an example.
        spec-type: optional-str
      example-number:
        description: |
          It shall be the example number.
        spec-type: int
    description: |
      This set of attributes specifies an example.
    mandatory-attributes: all
spec-name: Example Item Type
spec-type: spec
type: spec
```

#### 5.2.2.53 Test Case Item Type

This type refines the *Root Item Type* (page 25) through the `type` attribute if the value is `test-case`. This set of attributes specifies a test case. All explicit attributes shall be specified. The explicit attributes for this type are:

**test-actions**
The attribute value shall be a list. Each list element shall be a *Test Case Action* (page 107).

**test-brief**
The attribute value shall be a string. It shall be the test case brief description.

**test-context**
The attribute value shall be a list. Each list element shall be a *Test Context Member* (page 108).

**test-context-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test context support code. The context support code is placed at file scope before the test context definition.

**test-description**
The attribute value shall be an optional string. It shall be the test case description.

**test-header**
The attribute value shall be a *Test Header* (page 108).

**test-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include <...>`.

**test-local-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include "..."`.

**test-setup**
The attribute value shall be a *Test Support Method* (page 109).

**test-stop**
The attribute value shall be a *Test Support Method* (page 109).

**test-support**
The attribute value shall be an optional string. If the value is present, then it shall be the test case support code. The support code is placed at file scope before the test case code.

**test-target**
The attribute value shall be a string. It shall be the path to the generated target test case source file.

**test-teardown**
The attribute value shall be a *Test Support Method* (page 109).

#### 5.2.2.54 Test Suite Item Type

This type refines the following types:

- *Root Item Type* (page 25) through the `type` attribute if the value is `memory-benchmark`
- *Root Item Type* (page 25) through the `type` attribute if the value is `test-suite`

This set of attributes specifies a test suite. All explicit attributes shall be specified. The explicit attributes for this type are:

**test-brief**
The attribute value shall be a string. It shall be the test suite brief description.

**test-code**
The attribute value shall be a string. It shall be the test suite code. The test suite code is placed at file scope in the target source file.

**test-description**
The attribute value shall be an optional string. It shall be the test suite description.

**test-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include <...>`.

**test-local-includes**
The attribute value shall be a list of strings. It shall be a list of header files included via `#include "..."`.

**test-target**
The attribute value shall be a string. It shall be the path to the generated target test suite source file.

### 5.2.3 Specification Attribute Sets and Value Types

#### 5.2.3.1 Action Requirement Boolean Expression

A value of this type is a boolean expression.

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be an *Action Requirement Boolean Expression* (page 62). The *and* operator evaluates to the *logical and* of the evaluation results of the expressions in the list.

  **not**
  The attribute value shall be an *Action Requirement Boolean Expression* (page 62). The *not* operator evaluates to the *logical not* of the evaluation results of the expression.

  **or**
  The attribute value shall be a list. Each list element shall be an *Action Requirement Boolean Expression* (page 62). The *or* operator evaluates to the *logical or* of the evaluation results of the expressions in the list.

  **post-conditions**
  The attribute value shall be an *Action Requirement Expression Condition Set* (page 64). The *post-conditions* operator evaluates to true, if the post-condition states of the associated transition are contained in the specified post-condition set, otherwise to false.

  **pre-conditions**
  The attribute value shall be an *Action Requirement Expression Condition Set* (page 64). The *pre-conditions* operator evaluates to true, if the pre-condition states of the associated transition are contained in the specified pre-condition set, otherwise to false.

- The value may be a list. Each list element shall be an *Action Requirement Boolean Expression* (page 62). This list of expressions evaluates to the *logical or* of the evaluation results of the expressions in the list.

This type is used by the following types:

- *Action Requirement Boolean Expression* (page 62)
- *Action Requirement Expression* (page 63)

#### 5.2.3.2 Action Requirement Condition

This set of attributes defines an action pre-condition or post-condition. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be an *Action Requirement Name* (page 64).

**states**
The attribute value shall be a list. Each list element shall be an *Action Requirement State* (page 65).

**test-epilogue**
The attribute value shall be an optional string. If the value is present, then it shall be the test epilogue code. The epilogue code is placed in the test condition preparation or check before the state-specific code. The code may use a local variable `ctx` which points to the test context, see *Test Context Member* (page 108).

**test-prologue**
The attribute value shall be an optional string. If the value is present, then it shall be the test prologue code. The prologue code is placed in the test condition preparation or check after the state-specific code. The code may use a local variable `ctx` which points to the test context, see *Test Context Member* (page 108).

This type is used by the following types:

- *Action Requirement Item Type* (page 49)

#### 5.2.3.3 Action Requirement Expression

This set of attributes defines an expression which may define the state of a post-condition. The `else` and `specified-by` shall be used individually. The `if` and `then` or `then-specified-by` expressions shall be used together. At least one of the explicit attributes shall be specified. The explicit attributes for this type are:

**else**
The attribute value shall be an *Action Requirement Expression State Name* (page 64). It shall be the name of the state of the post-condition.

**if**
The attribute value shall be an *Action Requirement Boolean Expression* (page 62). If the boolean expression evaluates to true, then the state is defined according to the `then` attribute value.

**specified-by**
The attribute value shall be an *Action Requirement Name* (page 64). It shall be the name of a pre-condition. The name of the state of the pre-condition in the associated transition defines the name of the state of the post-condition.

**then**
The attribute value shall be an *Action Requirement Expression State Name* (page 64). It shall be the name of the state of the post-condition.

**then-specified-by**
The attribute value shall be an *Action Requirement Name* (page 64). It shall be the name of a pre-condition. The name of the state of the pre-condition in the associated transition defines the name of the state of the post-condition.

#### 5.2.3.4 Action Requirement Expression Condition Set

This set of attributes defines for the specified conditions a set of states. Generic attributes may be specified. Each generic attribute key shall be an *Action Requirement Name* (page 64). Each generic attribute value shall be an *Action Requirement Expression State Set* (page 64). There shall be at most one generic attribute key for each condition. The key name shall be the condition name. The value of each generic attribute shall be a set of states of the condition.

This type is used by the following types:

- *Action Requirement Boolean Expression* (page 62)

#### 5.2.3.5 Action Requirement Expression State Name

The value shall be a string. It shall be the name of a state of the condition or `N/A` if the condition is not applicable. The value

- shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`",
- or, shall be equal to "`N/A`".

This type is used by the following types:

- *Action Requirement Expression* (page 63)

#### 5.2.3.6 Action Requirement Expression State Set

A value of this type shall be of one of the following variants:

- The value may be a list. Each list element shall be an *Action Requirement Expression State Name* (page 64). The list defines a set of states of the condition.
- The value may be a string. It shall be the name of a state of the condition or `N/A` if the condition is not applicable. The value
  - shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`",
  - or, shall be equal to "`N/A`".

This type is used by the following types:

- *Action Requirement Expression Condition Set* (page 64)

#### 5.2.3.7 Action Requirement Name

The value shall be a string. It shall be the name of a condition or a state of a condition used to define pre-conditions and post-conditions of an action requirement. It shall be formatted in CamelCase. It should be brief and abbreviated. The rationale for this is that the names are used in tables and the horizontal space is limited by the page width. The more conditions you have in an action requirement, the shorter the names should be. The name `NA` is reserved and indicates that a condition is not applicable. The value

- shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`",
- and, shall be not equal to "`NA`".

This type is used by the following types:

- *Action Requirement Condition* (page 63)
- *Action Requirement Expression Condition Set* (page 64)

- *Action Requirement Expression* (page 63)
- *Action Requirement Skip Reasons* (page 65)
- *Action Requirement State* (page 65)
- *Action Requirement Transition Post-Conditions* (page 66)
- *Action Requirement Transition Pre-Conditions* (page 67)

#### 5.2.3.8 Action Requirement Skip Reasons

This set of attributes specifies skip reasons used to justify why transitions in the transition map are skipped. Generic attributes may be specified. Each generic attribute key shall be an *Action Requirement Name* (page 64). Each generic attribute value shall be a string. The key defines the name of a skip reason. The name can be used in *Action Requirement Transition Post-Conditions* (page 66) to skip the corresponding transitions. The value shall give a reason why the transitions are skipped.

This type is used by the following types:

- *Action Requirement Item Type* (page 49)

#### 5.2.3.9 Action Requirement State

This set of attributes defines an action pre-condition or post-condition state. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be an *Action Requirement Name* (page 64).

**test-code**
The attribute value shall be a string. It shall be the test code to prepare or check the state of the condition. The code may use a local variable `ctx` which points to the test context, see *Test Context Member* (page 108).

**text**
The attribute value shall be a *Requirement Text* (page 95). It shall define the state of the condition.

This type is used by the following types:

- *Action Requirement Condition* (page 63)

#### 5.2.3.10 Action Requirement Transition

This set of attributes defines the transition from multiple sets of states of pre-conditions to a set of states of post-conditions through an action in an action requirement. The ability to specify multiple sets of states of pre-conditions which result in a common set of post-conditions may allow a more compact specification of the transition map. For example, let us suppose you want to specify the action of a function with a pointer parameter. The function performs an early check that the pointer is NULL and in this case returns an error code. The pointer condition dominates the action outcome if the pointer is NULL. Other pre-condition states can be simply set to `all` for this transition. All explicit attributes shall be specified. The explicit attributes for this type are:

**enabled-by**
The attribute value shall be an *Enabled-By Expression* (page 77). The transition map may be

customized to support configuration variants through this attribute. The default transitions (`enabled-by: true`) shall be specified before the customized variants in the list.

**post-conditions**
The attribute value shall be an *Action Requirement Transition Post-Conditions* (page 66).

**pre-conditions**
The attribute value shall be an *Action Requirement Transition Pre-Conditions* (page 67).

This type is used by the following types:

- *Action Requirement Item Type* (page 49)

#### 5.2.3.11 Action Requirement Transition Post-Condition State

A value of this type shall be of one of the following variants:

- The value may be a list. Each list element shall be an *Action Requirement Expression* (page 63). The list contains expressions to define the state of the corresponding post-condition.
- The value may be a string. It shall be the name of a state of the corresponding post-condition or `N/A` if the post-condition is not applicable. The value
  - shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`",
  - or, shall be equal to "`N/A`".

This type is used by the following types:

- *Action Requirement Transition Post-Conditions* (page 66)

#### 5.2.3.12 Action Requirement Transition Post-Conditions

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes defines for each post-condition the state after the action for a transition in an action requirement. Generic attributes may be specified. Each generic attribute key shall be an *Action Requirement Name* (page 64). Each generic attribute value shall be an *Action Requirement Transition Post-Condition State* (page 66). There shall be exactly one generic attribute key for each post-condition. The key name shall be the post-condition name. The value of each generic attribute shall be the state of the post-condition or `N/A` if the post-condition is not applicable.
- The value may be a string. It shall be the name of a skip reason. If a skip reason is given instead of a listing of post-condition states, then this transition is skipped and no test code runs for this transition. The value
  - shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`",
  - and, shall be not equal to "`NA`".

This type is used by the following types:

- *Action Requirement Transition* (page 65)

#### 5.2.3.13 Action Requirement Transition Pre-Condition State Set

A value of this type shall be of one of the following variants:

- The value may be a list. Each list element shall be an *Action Requirement Name* (page 64). The list defines the set of states of the pre-condition in the transition.
- The value may be a string. The value `all` represents all states of the pre-condition in this transition. The value `N/A` marks the pre-condition as not applicable in this transition. The value shall be an element of
  - "`all`", and
  - "`N/A`".

This type is used by the following types:

- *Action Requirement Transition Pre-Conditions* (page 67)

#### 5.2.3.14 Action Requirement Transition Pre-Conditions

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes defines for each pre-condition the set of states before the action for a transition in an action requirement. Generic attributes may be specified. Each generic attribute key shall be an *Action Requirement Name* (page 64). Each generic attribute value shall be an *Action Requirement Transition Pre-Condition State Set* (page 67). There shall be exactly one generic attribute key for each pre-condition. The key name shall be the pre-condition name. The value of each generic attribute shall be a set of states of the pre-condition.
- The value may be a string. If this name is specified instead of explicit pre-condition states, then the post-condition states of this entry are used to define all remaining transitions of the map. The value shall be equal to "`default`".

This type is used by the following types:

- *Action Requirement Transition* (page 65)

#### 5.2.3.15 Application Configuration Option Name

The value shall be a string. It shall be the name of an application configuration option. The value shall match with the regular expression "`^(CONFIGURE_|BSP_)[A-Z0-9_]+$`".

This type is used by the following types:

- *Application Configuration Option Item Type* (page 41)

#### 5.2.3.16 Boolean or Integer or String

A value of this type shall be of one of the following variants:

- The value may be a boolean.
- The value may be an integer number.
- The value may be a string.

This type is used by the following types:

- *Build Option Action* (page 71)

- *Interface Return Value* (page 89)

#### 5.2.3.17 Build Assembler Option

The value shall be a string. It shall be an option for the assembler. The options are used to assemble the sources of this item. The options defined by this attribute succeed the options presented to the item by the build item context.

This type is used by the following types:

- *Build Script Item Type* (page 35)
- *Build Start File Item Type* (page 37)

#### 5.2.3.18 Build C Compiler Option

The value shall be a string. It shall be an option for the C compiler. The options are used to compile the sources of this item. The options defined by this attribute succeed the options presented to the item by the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build BSP Item Type* (page 28)
- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Option C Compiler Check Action* (page 74)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.19 Build C Preprocessor Option

The value shall be a string. It shall be an option for the C preprocessor. The options are used to preprocess the sources of this item. The options defined by this attribute succeed the options presented to the item by the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build BSP Item Type* (page 28)
- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Script Item Type* (page 35)
- *Build Start File Item Type* (page 37)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.20 Build C++ Compiler Option

The value shall be a string. It shall be an option for the C++ compiler. The options are used to compile the sources of this item. The options defined by this attribute succeed the options presented to the item by the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Option C++ Compiler Check Action* (page 74)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.21 Build Dependency Conditional Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `build-dependency-conditional`. It defines the build dependency conditional role of links. All explicit attributes shall be specified. The explicit attributes for this type are:

**enabled-by**
The attribute value shall be an *Enabled-By Expression* (page 77). It shall define under which conditions the build dependency is enabled.

#### 5.2.3.22 Build Dependency Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `build-dependency`. It defines the build dependency role of links.

#### 5.2.3.23 Build Include Path

The value shall be a string. It shall be a path to header files. The path is used by the C preprocessor to search for header files. It succeeds the includes presented to the item by the build item context. For an *Build Group Item Type* (page 31) item the includes are visible to all items referenced by the group item. For *Build BSP Item Type* (page 28), *Build Objects Item Type* (page 33), *Build Library Item Type* (page 32), *Build Start File Item Type* (page 37), and *Build Test Program Item Type* (page 38) items the includes are only visible to the sources specified by the item itself and they do not propagate to referenced items.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build BSP Item Type* (page 28)
- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Script Item Type* (page 35)

- *Build Start File Item Type* (page 37)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.24 Build Install Directive

This set of attributes specifies files installed by a build item. All explicit attributes shall be specified. The explicit attributes for this type are:

**destination**
The attribute value shall be a string. It shall be the install destination directory.

**source**
The attribute value shall be a list of strings. It shall be the list of source files to be installed in the destination directory. The path to a source file shall be relative to the directory of the `wscript`.

This type is used by the following types:

- *Build BSP Item Type* (page 28)
- *Build Group Item Type* (page 31)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)

#### 5.2.3.25 Build Install Path

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It shall be the installation path of a *Build Target* (page 76).

This type is used by the following types:

- *Build Configuration File Item Type* (page 30)
- *Build Configuration Header Item Type* (page 30)
- *Build Library Item Type* (page 32)
- *Build Start File Item Type* (page 37)

#### 5.2.3.26 Build Link Static Library Directive

The value shall be a string. It shall be an external static library identifier. The library is used to link programs referenced by this item, e.g. `m` for `libm.a`. The library is added to the build command through the `stlib` attribute. It shall not be used for internal static libraries. Internal static libraries shall be specified through the `use-after` and `use-before` attributes to enable a proper build dependency tracking.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.27 Build Linker Option

The value shall be a string. It shall be an option for the linker. The options are used to link executables. The options defined by this attribute succeed the options presented to the item by the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.28 Build Option Action

This set of attributes specifies a build option action. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

**append-test-cppflags**
The attribute value shall be a string. It shall be the name of a test program. The action appends the action value to the `CPPFLAGS` of the test program. The name shall correspond to the name of a *Build Test Program Item Type* (page 38) item. Due to the processing order of items, there is no way to check if the name specified by the attribute value is valid.

**assert-aligned**
The attribute value shall be an integer number. The action asserts that the action value is aligned according to the attribute value.

**assert-eq**
The attribute value shall be a *Boolean or Integer or String* (page 67). The action asserts that the action value is equal to the attribute value.

**assert-ge**
The attribute value shall be an *Integer or String* (page 80). The action asserts that the action value is greater than or equal to the attribute value.

**assert-gt**
The attribute value shall be an *Integer or String* (page 80). The action asserts that the action value is greater than the attribute value.

**assert-in-set**
The attribute value shall be a list. Each list element shall be an *Integer or String* (page 80). The action asserts that the action value is in the attribute value set.

**assert-int16**
The attribute shall have no value. The action asserts that the action value is a valid signed 16-bit integer.

**assert-int32**
The attribute shall have no value. The action asserts that the action value is a valid signed 32-bit integer.

**assert-int64**
The attribute shall have no value. The action asserts that the action value is a valid signed 64-bit integer.

**assert-int8**
The attribute shall have no value. The action asserts that the action value is a valid signed 8-bit integer.

**assert-le**
The attribute value shall be an *Integer or String* (page 80). The action asserts that the action value is less than or equal to the attribute value.

**assert-lt**
The attribute value shall be an *Integer or String* (page 80). The action asserts that the action value is less than the attribute value.

**assert-ne**
The attribute value shall be a *Boolean or Integer or String* (page 67). The action asserts that the action value is not equal to the attribute value.

**assert-power-of-two**
The attribute shall have no value. The action asserts that the action value is a power of two.

**assert-uint16**
The attribute shall have no value. The action asserts that the action value is a valid unsigned 16-bit integer.

**assert-uint32**
The attribute shall have no value. The action asserts that the action value is a valid unsigned 32-bit integer.

**assert-uint64**
The attribute shall have no value. The action asserts that the action value is a valid unsigned 64-bit integer.

**assert-uint8**
The attribute shall have no value. The action asserts that the action value is a valid unsigned 8-bit integer.

**check-cc**
The attribute value shall be a *Build Option C Compiler Check Action* (page 74).

**check-cxx**
The attribute value shall be a *Build Option C++ Compiler Check Action* (page 74).

**comment**
The attribute value shall be a string. There is no action performed. The attribute value is a comment.

**define**
The attribute value shall be an optional string. The action adds a define to the configuration set. If the attribute value is present, then it is used as the name of the define, otherwise the `name` of the item is used. The value of the define is the action value. If the action value is a string, then it is quoted.

**define-condition**
The attribute value shall be an optional string. The action adds a conditional define to the configuration set. If the attribute value is present, then it is used as the name of the define, otherwise the `name` of the item is used. The value of the define is the action value.

**define-unquoted**
The attribute value shall be an optional string. The action adds a define to the configuration

set. If the attribute value is present, then it is used as the name of the define, otherwise the `name` of the item is used. The value of the define is the action value. If the action value is a string, then it is not quoted.

**env-append**
The attribute value shall be an optional string. The action appends the action value to an environment of the configuration set. If the attribute value is present, then it is used as the name of the environment variable, otherwise the `name` of the item is used.

**env-assign**
The attribute value shall be an optional string. The action assigns the action value to an environment of the configuration set. If the attribute value is present, then it is used as the name of the environment variable, otherwise the `name` of the item is used.

**env-enable**
The attribute value shall be an optional string. If the action value is true, then a name is appended to the `ENABLE` environment variable of the configuration set. If the attribute value is present, then it is used as the name, otherwise the `name` of the item is used.

**find-program**
The attribute shall have no value. The action tries to find the program specified by the action value. Uses the `${PATH}` to find the program. Returns the result of the find operation, e.g. a path to the program.

**find-tool**
The attribute shall have no value. The action tries to find the tool specified by the action value. Uses the tool paths specified by the `--rtems-tools` command line option. Returns the result of the find operation, e.g. a path to the program.

**format-and-define**
The attribute value shall be an optional string. The action adds a define to the configuration set. If the attribute value is present, then it is used as the name of the define, otherwise the `name` of the item is used. The value of the define is the action value. The value is formatted according to the `format` attribute value.

**get-boolean**
The attribute shall have no value. The action gets the action value for subsequent actions from a configuration file variable named by the items `name` attribute. If no such variable exists in the configuration file, then the default value is used. The value is converted to a boolean.

**get-env**
The attribute value shall be a string. The action gets the action value for subsequent actions from the environment variable of the configuration set named by the attribute value.

**get-integer**
The attribute shall have no value. The action gets the action value for subsequent actions from a configuration file variable named by the items `name` attribute. If no such variable exists in the configuration file, then the default value is used. The value is converted to an integer.

**get-string**
The attribute shall have no value. The action gets the action value for subsequent actions from a configuration file variable named by the items `name` attribute. If no such variable exists in the configuration file, then the default value is used. The value is converted to a string.

**get-string-command-line**
The attribute value shall be a string. The action gets the action value for subsequent actions from the value of a command line option named by the items `name` attribute. If no such

command line option is present, then the attribute value is used. The value is converted to a string.

**script**
The attribute value shall be a string. The action executes the attribute value with the Python `eval()` function in the context of the script action handler.

**set-test-state**
The attribute value shall be a *Build Option Set Test State Action* (page 75).

**set-value**
The attribute value may have any type. The action sets the action value for subsequent actions to the attribute value.

**set-value-enabled-by**
The attribute value shall be a list. Each list element shall be a *Build Option Value* (page 75). The action sets the action value for subsequent actions to the first enabled attribute value.

**split**
The attribute shall have no value. The action splits the action value.

**substitute**
The attribute shall have no value. The action performs a `${VARIABLE}` substitution on the action value. Use `$$` for a plain `$` character.

This type is used by the following types:

- *Build Option Item Type* (page 34)

#### 5.2.3.29 Build Option C Compiler Check Action

This set of attributes specifies a check done using the C compiler. All explicit attributes shall be specified. The explicit attributes for this type are:

**cflags**
The attribute value shall be a list. Each list element shall be a *Build C Compiler Option* (page 68).

**fragment**
The attribute value shall be a string. It shall be a code fragment used to check the availability of a certain feature through compilation with the C compiler. The resulting object is not linked to an executable.

**message**
The attribute value shall be a string. It shall be a description of the feature to check.

This type is used by the following types:

- *Build Option Action* (page 71)

#### 5.2.3.30 Build Option C++ Compiler Check Action

This set of attributes specifies a check done using the C++ compiler. All explicit attributes shall be specified. The explicit attributes for this type are:

**cxxflags**
The attribute value shall be a list. Each list element shall be a *Build C++ Compiler Option* (page 69).

**fragment**
The attribute value shall be a string. It shall be a code fragment used to check the availability of a certain feature through compilation with the C++ compiler. The resulting object is not linked to an executable.

**message**
The attribute value shall be a string. It shall be a description of the feature to check.

This type is used by the following types:

- *Build Option Action* (page 71)

#### 5.2.3.31 Build Option Name

The value shall be a string. It shall be the name of the build option. The value shall match with the regular expression "`^[a-zA-Z_][a-zA-Z0-9_]*$`".

This type is used by the following types:

- *Build Option Item Type* (page 34)

#### 5.2.3.32 Build Option Set Test State Action

This set of attributes specifies the test state for a set of test programs with an optional reason. All explicit attributes shall be specified. The explicit attributes for this type are:

**reason**
The attribute value shall be an optional string. If the value is present, then it shall be the reason for the test state definition.

**state**
The attribute value shall be a *Build Test State* (page 76). It shall be the test state for the associated list of tests.

**tests**
The attribute value shall be a list of strings. It shall be the list of test program names associated with the test state. The names shall correspond to the name of a *Build Test Program Item Type* (page 38) or *Build Ada Test Program Item Type* (page 27) item. Due to the processing order of items, there is no way to check if a specified test program name is valid.

This type is used by the following types:

- *Build Option Action* (page 71)

#### 5.2.3.33 Build Option Value

This set of attributes specifies an optional build option value. All explicit attributes shall be specified. The explicit attributes for this type are:

**enabled-by**
The attribute value shall be an *Enabled-By Expression* (page 77).

**value**
The attribute value may have any type. If the associated enabled-by expression evaluates to true for the current enabled set, then the attribute value is active and may get selected.

This type is used by the following types:

- *Build Option Action* (page 71)

- *Build Option Item Type* (page 34)

#### 5.2.3.34 Build Source

The value shall be a string. It shall be a source file. The path to a source file shall be relative to the directory of the `wscript`.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build BSP Item Type* (page 28)
- *Build Library Item Type* (page 32)
- *Build Objects Item Type* (page 33)
- *Build Start File Item Type* (page 37)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.35 Build Target

The value shall be a string. It shall be the target file path. The path to the target file shall be relative to the directory of the `wscript`. The target file is located in the build tree.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Configuration File Item Type* (page 30)
- *Build Configuration Header Item Type* (page 30)
- *Build Library Item Type* (page 32)
- *Build Script Item Type* (page 35)
- *Build Start File Item Type* (page 37)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.36 Build Test State

The value shall be a string. This string defines a test state. The value shall be an element of

- “`benchmark`”,
- “`exclude`”,
- “`expected-fail`”,
- “`indeterminate`”, and
- “`user-input`”.

This type is used by the following types:

- *Build Option Set Test State Action* (page 75)

#### 5.2.3.37 Build Use After Directive

The value shall be a string. It shall be an internal static library identifier. The library is used to link programs referenced by this item, e.g. `z` for `libz.a`. The library is placed after the use items of the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Group Item Type* (page 31)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.38 Build Use Before Directive

The value shall be a string. It shall be an internal static library identifier. The library is used to link programs referenced by this item, e.g. `z` for `libz.a`. The library is placed before the use items of the build item context.

This type is used by the following types:

- *Build Ada Test Program Item Type* (page 27)
- *Build Group Item Type* (page 31)
- *Build Script Item Type* (page 35)
- *Build Test Program Item Type* (page 38)

#### 5.2.3.39 Constraint Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `constraint`. It defines the constraint role of links. The link target shall be a constraint.

#### 5.2.3.40 Copyright

The value shall be a string. It shall be a copyright statement of a copyright holder of the specification item. The value

- shall match with the regular expression "`^\s*Copyright\s+\(C\)\s+[0-9]+,\s*[0-9]+\s+.+\s*$`",
- or, shall match with the regular expression "`^\s*Copyright\s+\(C\)\s+[0-9]+\s+.+\s*$`",
- or, shall match with the regular expression "`^\s*Copyright\s+\(C\)\s+.+\s*$`".

This type is used by the following types:

- *Root Item Type* (page 25)

#### 5.2.3.41 Enabled-By Expression

A value of this type shall be an expression which defines under which conditions the specification item or parts of it are enabled. The expression is evaluated with the use of an *enabled set*. This is a set of strings which indicate enabled features.

A value of this type shall be of one of the following variants:

- The value may be a boolean. This expression evaluates directly to the boolean value.
- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be an *Enabled-By Expression* (page 77). The *and* operator evaluates to the *logical and* of the evaluation results of the expressions in the list.

  **not**
  The attribute value shall be an *Enabled-By Expression* (page 77). The *not* operator evaluates to the *logical not* of the evaluation results of the expression.

  **or**
  The attribute value shall be a list. Each list element shall be an *Enabled-By Expression* (page 77). The *or* operator evaluates to the *logical or* of the evaluation results of the expressions in the list.
- The value may be a list. Each list element shall be an *Enabled-By Expression* (page 77). This list of expressions evaluates to the *logical or* of the evaluation results of the expressions in the list.
- The value may be a string. If the value is in the *enabled set*, this expression evaluates to true, otherwise to false.

This type is used by the following types:

- *Action Requirement Transition* (page 65)
- *Build Dependency Conditional Link Role* (page 69)
- *Build Option Value* (page 75)
- *Enabled-By Expression* (page 77)
- *Interface Include Link Role* (page 87)
- *Root Item Type* (page 25)

Please have a look at the following example:

```
enabled-by:
  and:
  - RTEMS_NETWORKING
  - not: RTEMS_SMP
```

#### 5.2.3.42 External Document Reference

This type refines the *External Reference* (page 79) through the `type` attribute if the value is `document`. It specifies a reference to a document.

All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be the name of the document.

#### 5.2.3.43 External File Reference

This type refines the *External Reference* (page 79) through the `type` attribute if the value is `file`. It specifies a reference to a file.

All explicit attributes shall be specified. The explicit attributes for this type are:

**hash**
The attribute value shall be a *SHA256 Hash Value* (page 99). It shall be the SHA256 hash value of the content of the referenced file.

#### 5.2.3.44 External Reference

This set of attributes specifies a reference to some object external to the specification. All explicit attributes shall be specified. The explicit attributes for this type are:

**identifier**
The attribute value shall be a string. It shall be the type-specific identifier of the referenced object. For *group* references use the Doxygen group identifier. For *file* references use a file system path to the file.

**type**
The attribute value shall be a *Name* (page 90). It shall be the type of the referenced object.

This type is refined by the following types:

- *External Document Reference* (page 78)
- *External File Reference* (page 79)
- *Generic External Reference* (page 79)

This type is used by the following types:

- *Interface Unspecified Header File Item Type* (page 46)
- *Interface Unspecified Item Type* (page 46)
- *Requirement Item Type* (page 48)
- *Requirement Validation Item Type* (page 57)

#### 5.2.3.45 Function Implementation Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `function-implementation`. It defines the function implementation role of links. It is used to indicate that a *Functional Requirement Item Type* (page 49) item specifies parts of the function.

#### 5.2.3.46 Generic External Reference

This type refines the following types:

- *External Reference* (page 79) through the `type` attribute if the value is `define`
- *External Reference* (page 79) through the `type` attribute if the value is `function`
- *External Reference* (page 79) through the `type` attribute if the value is `group`
- *External Reference* (page 79) through the `type` attribute if the value is `macro`
- *External Reference* (page 79) through the `type` attribute if the value is `url`

- *External Reference* (page 79) through the `type` attribute if the value is `variable`

It specifies a reference to an object of the specified type.

#### 5.2.3.47 Glossary Membership Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `glossary-member`. It defines the glossary membership role of links.

#### 5.2.3.48 Integer or String

A value of this type shall be of one of the following variants:

- The value may be an integer number.
- The value may be a string.

This type is used by the following types:

- *Application Configuration Value Option Item Type* (page 42)
- *Build Option Action* (page 71)

#### 5.2.3.49 Interface Brief Description

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It shall be the brief description of the interface. It should be a single sentence. The value shall not match with the regular expression "`\n\n`".

This type is used by the following types:

- *Interface Compound Item Type* (page 42)
- *Interface Compound Member Definition* (page 81)
- *Interface Define Item Type* (page 43)
- *Interface Enum Item Type* (page 43)
- *Interface Enumerator Item Type* (page 44)
- *Interface Function or Macro Item Type* (page 44)
- *Interface Group Item Type* (page 45)
- *Interface Header File Item Type* (page 45)
- *Interface Typedef Item Type* (page 45)
- *Interface Variable Item Type* (page 47)
- *Register Bits Definition* (page 92)
- *Register Block Item Type* (page 47)
- *Register Definition* (page 94)

#### 5.2.3.50 Interface Compound Definition Kind

The value shall be a string. It specifies how the interface compound is defined. It may be a typedef only, the struct or union only, or a typedef with a struct or union definition. The value shall be an element of

- "`struct-only`",
- "`typedef-and-struct`",
- "`typedef-and-union`",
- "`typedef-only`", and
- "`union-only`".

This type is used by the following types:

- *Interface Compound Item Type* (page 42)

#### 5.2.3.51 Interface Compound Member Compound

This type refines the following types:

- *Interface Compound Member Definition* (page 81) through the `kind` attribute if the value is `struct`
- *Interface Compound Member Definition* (page 81) through the `kind` attribute if the value is `union`

This set of attributes specifies an interface compound member compound. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be a list. Each list element shall be an *Interface Compound Member Definition Directive* (page 82).

#### 5.2.3.52 Interface Compound Member Declaration

This type refines the *Interface Compound Member Definition* (page 81) through the `kind` attribute if the value is `member`. This set of attributes specifies an interface compound member declaration. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be a string. It shall be the interface compound member declaration. On the declaration a context-sensitive substitution of item variables is performed.

#### 5.2.3.53 Interface Compound Member Definition

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies an interface compound member definition. All explicit attributes shall be specified. The explicit attributes for this type are:

  **brief**
  The attribute value shall be an *Interface Brief Description* (page 80).

  **description**
  The attribute value shall be an *Interface Description* (page 83).

**kind**
The attribute value shall be a string. It shall be the interface compound member kind.

**name**
The attribute value shall be a string. It shall be the interface compound member name.

- There may be no value (null).

This type is refined by the following types:

- *Interface Compound Member Compound* (page 81)
- *Interface Compound Member Declaration* (page 81)

This type is used by the following types:

- *Interface Compound Member Definition Directive* (page 82)
- *Interface Compound Member Definition Variant* (page 82)

#### 5.2.3.54 Interface Compound Member Definition Directive

This set of attributes specifies an interface compound member definition directive. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be an *Interface Compound Member Definition* (page 81). The default definition will be used if no variant-specific definition is enabled.

**variants**
The attribute value shall be a list. Each list element shall be an *Interface Compound Member Definition Variant* (page 82).

This type is used by the following types:

- *Interface Compound Item Type* (page 42)
- *Interface Compound Member Compound* (page 81)

#### 5.2.3.55 Interface Compound Member Definition Variant

This set of attributes specifies an interface compound member definition variant. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be an *Interface Compound Member Definition* (page 81). The definition will be used if the expression defined by the `enabled-by` attribute evaluates to true. In generated header files, the expression is evaluated by the C preprocessor.

**enabled-by**
The attribute value shall be an *Interface Enabled-By Expression* (page 84).

This type is used by the following types:

- *Interface Compound Member Definition Directive* (page 82)

#### 5.2.3.56 Interface Definition

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It shall be the definition. On the definition a context-sensitive substitution of item variables is performed.

This type is used by the following types:

- *Interface Definition Directive* (page 83)
- *Interface Definition Variant* (page 83)

#### 5.2.3.57 Interface Definition Directive

This set of attributes specifies an interface definition directive. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be an *Interface Definition* (page 83). The default definition will be used if no variant-specific definition is enabled.

**variants**
The attribute value shall be a list. Each list element shall be an *Interface Definition Variant* (page 83).

This type is used by the following types:

- *Interface Define Item Type* (page 43)
- *Interface Enumerator Item Type* (page 44)
- *Interface Typedef Item Type* (page 45)
- *Interface Variable Item Type* (page 47)

#### 5.2.3.58 Interface Definition Variant

This set of attributes specifies an interface definition variant. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be an *Interface Definition* (page 83). The definition will be used if the expression defined by the `enabled-by` attribute evaluates to true. In generated header files, the expression is evaluated by the C preprocessor.

**enabled-by**
The attribute value shall be an *Interface Enabled-By Expression* (page 84).

This type is used by the following types:

- *Interface Definition Directive* (page 83)

#### 5.2.3.59 Interface Description

A value of this type shall be of one of the following variants:

- There may be no value (null).

- The value may be a string. It shall be the description of the interface. The description should be short and concentrate on the average case. All special cases, usage notes, constraints, error conditions, configuration dependencies, references, etc. should be described in the *Interface Notes* (page 87).

This type is used by the following types:

- *Application Configuration Option Item Type* (page 41)
- *Interface Compound Item Type* (page 42)
- *Interface Compound Member Definition* (page 81)
- *Interface Define Item Type* (page 43)
- *Interface Enum Item Type* (page 43)
- *Interface Enumerator Item Type* (page 44)
- *Interface Function or Macro Item Type* (page 44)
- *Interface Group Item Type* (page 45)
- *Interface Parameter* (page 88)
- *Interface Return Value* (page 89)
- *Interface Typedef Item Type* (page 45)
- *Interface Variable Item Type* (page 47)
- *Register Bits Definition* (page 92)
- *Register Block Item Type* (page 47)
- *Register Definition* (page 94)

#### 5.2.3.60 Interface Enabled-By Expression

A value of this type shall be an expression which defines under which conditions an interface definition is enabled. In generated header files, the expression is evaluated by the C preprocessor.

A value of this type shall be of one of the following variants:

- The value may be a boolean. It is converted to 0 or 1. It defines a symbol in the expression.
- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be an *Interface Enabled-By Expression* (page 84). The *and* operator defines a *logical and* of the expressions in the list.

  **not**
  The attribute value shall be an *Interface Enabled-By Expression* (page 84). The *not* operator defines a *logical not* of the expression.

  **or**
  The attribute value shall be a list. Each list element shall be an *Interface Enabled-By Expression* (page 84). The *or* operator defines a *logical or* of the expressions in the list.

- The value may be a list. Each list element shall be an *Interface Enabled-By Expression* (page 84). It defines a *logical or* of the expressions in the list.
- The value may be a string. It defines a symbol in the expression.

This type is used by the following types:

- *Interface Compound Member Definition Variant* (page 82)
- *Interface Definition Variant* (page 83)
- *Interface Enabled-By Expression* (page 84)
- *Interface Function or Macro Definition Variant* (page 86)
- *Register Bits Definition Variant* (page 93)
- *Register Block Member Definition Variant* (page 94)

#### 5.2.3.61 Interface Enum Definition Kind

The value shall be a string. It specifies how the enum is defined. It may be a typedef only, the enum only, or a typedef with an enum definition. The value shall be an element of

- "`enum-only`",
- "`typedef-and-enum`", and
- "`typedef-only`".

This type is used by the following types:

- *Interface Enum Item Type* (page 43)

#### 5.2.3.62 Interface Enumerator Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-enumerator`. It defines the interface enumerator role of links.

#### 5.2.3.63 Interface Function Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-function`. It defines the interface function role of links. It is used to indicate that a *Action Requirement Item Type* (page 49) item specifies functional requirements of an *Interface Function or Macro Item Type* (page 44) item.

#### 5.2.3.64 Interface Function or Macro Definition

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies a function definition. All explicit attributes shall be specified. The explicit attributes for this type are:

  **attributes**
  The attribute value shall be an optional string. If the value is present, then it shall be the function attributes. On the attributes a context-sensitive substitution of item variables is performed. A function attribute is for example the indication that the function does not return to the caller.

**body**
The attribute value shall be an optional string. If the value is present, then it shall be the definition of a static inline function. On the function definition a context-sensitive substitution of item variables is performed. If no value is present, then the function is declared as an external function.

**params**
The attribute value shall be a list of strings. It shall be the list of parameter declarations of the function. On the function parameter declarations a context-sensitive substitution of item variables is performed.

**return**
The attribute value shall be an optional string. If the value is present, then it shall be the function return type. On the return type a context-sensitive substitution of item variables is performed.

- There may be no value (null).

This type is used by the following types:

- *Interface Function or Macro Definition Directive* (page 86)
- *Interface Function or Macro Definition Variant* (page 86)

#### 5.2.3.65 Interface Function or Macro Definition Directive

This set of attributes specifies a function or macro definition directive. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be an *Interface Function or Macro Definition* (page 85). The default definition will be used if no variant-specific definition is enabled.

**variants**
The attribute value shall be a list. Each list element shall be an *Interface Function or Macro Definition Variant* (page 86).

This type is used by the following types:

- *Interface Function or Macro Item Type* (page 44)

#### 5.2.3.66 Interface Function or Macro Definition Variant

This set of attributes specifies a function or macro definition variant. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be an *Interface Function or Macro Definition* (page 85). The definition will be used if the expression defined by the `enabled-by` attribute evaluates to true. In generated header files, the expression is evaluated by the C preprocessor.

**enabled-by**
The attribute value shall be an *Interface Enabled-By Expression* (page 84).

This type is used by the following types:

- *Interface Function or Macro Definition Directive* (page 86)

#### 5.2.3.67 Interface Group Identifier

The value shall be a string. It shall be the identifier of the interface group. The value shall match with the regular expression "`^[A-Z][a-zA-Z0-9]*$`".

This type is used by the following types:

- *Interface Group Item Type* (page 45)
- *Register Block Item Type* (page 47)

#### 5.2.3.68 Interface Group Membership Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-ingroup`. It defines the interface group membership role of links.

#### 5.2.3.69 Interface Hidden Group Membership Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-ingroup-hidden`. It defines the interface hidden group membership role of links. This role may be used to make an interface a group member and hide this relationship in the documentation. An example is an optimized macro implementation of a directive which has the same name as the corresponding directive.

#### 5.2.3.70 Interface Include Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-include`. It defines the interface include role of links and is used to indicate that an interface container includes another interface container. For example, one header file includes another header file. All explicit attributes shall be specified. The explicit attributes for this type are:

**enabled-by**
: The attribute value shall be an *Enabled-By Expression* (page 77). It shall define under which conditions the interface container is included.

#### 5.2.3.71 Interface Notes

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It shall be the notes for the interface.

This type is used by the following types:

- *Application Configuration Option Item Type* (page 41)
- *Interface Compound Item Type* (page 42)
- *Interface Define Item Type* (page 43)
- *Interface Enumerator Item Type* (page 44)
- *Interface Function or Macro Item Type* (page 44)
- *Interface Typedef Item Type* (page 45)
- *Interface Variable Item Type* (page 47)
- *Register Block Item Type* (page 47)

#### 5.2.3.72 Interface Parameter

This set of attributes specifies an interface parameter. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be an *Interface Description* (page 83).

**dir**
The attribute value shall be an *Interface Parameter Direction* (page 88).

**name**
The attribute value shall be a string. It shall be the interface parameter name.

This type is used by the following types:

- *Interface Function or Macro Item Type* (page 44)
- *Interface Typedef Item Type* (page 45)

#### 5.2.3.73 Interface Parameter Direction

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It specifies the interface parameter direction. The value shall be an element of
  - "`in`",
  - "`out`", and
  - "`inout`".

This type is used by the following types:

- *Interface Parameter* (page 88)
- *Test Run Parameter* (page 109)

#### 5.2.3.74 Interface Placement Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-placement`. It defines the interface placement role of links. It is used to indicate that an interface definition is placed into an interface container, for example a header file.

#### 5.2.3.75 Interface Return Directive

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies an interface return. All explicit attributes shall be specified. The explicit attributes for this type are:

  **return**
  The attribute value shall be an optional string. It shall describe the interface return for unspecified return values.

  **return-values**
  The attribute value shall be a list. Each list element shall be an *Interface Return Value* (page 89).

- There may be no value (null).

This type is used by the following types:

- *Interface Function or Macro Item Type* (page 44)
- *Interface Typedef Item Type* (page 45)

#### 5.2.3.76 Interface Return Value

This set of attributes specifies an interface return value. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be an *Interface Description* (page 83).

**value**
The attribute value shall be a *Boolean or Integer or String* (page 67). It shall be the described interface return value.

This type is used by the following types:

- *Interface Return Directive* (page 88)

#### 5.2.3.77 Interface Target Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `interface-target`. It defines the interface target role of links. It is used for interface forward declarations.

#### 5.2.3.78 Link

This set of attributes specifies a link from one specification item to another specification item. The links in a list are ordered. The first link in the list is processed first. All explicit attributes shall be specified. The explicit attributes for this type are:

**role**
The attribute value shall be a *Name* (page 90). It shall be the role of the link.

**uid**
The attribute value shall be an *UID* (page 110). It shall be the absolute or relative UID of the link target item.

This type is refined by the following types:

- *Build Dependency Conditional Link Role* (page 69)
- *Build Dependency Link Role* (page 69)
- *Constraint Link Role* (page 77)
- *Function Implementation Link Role* (page 79)
- *Glossary Membership Link Role* (page 80)
- *Interface Enumerator Link Role* (page 85)
- *Interface Function Link Role* (page 85)
- *Interface Group Membership Link Role* (page 87)
- *Interface Hidden Group Membership Link Role* (page 87)

- *Interface Include Link Role* (page 87)
- *Interface Placement Link Role* (page 88)
- *Interface Target Link Role* (page 89)
- *Performance Runtime Limits Link Role* (page 91)
- *Placement Order Link Role* (page 91)
- *Proxy Member Link Role* (page 92)
- *Register Block Include Role* (page 93)
- *Requirement Refinement Link Role* (page 95)
- *Requirement Validation Link Role* (page 97)
- *Runtime Measurement Request Link Role* (page 98)
- *Specification Member Link Role* (page 105)
- *Specification Refinement Link Role* (page 105)
- *Unit Test Link Role* (page 110)

This type is used by the following types:

- *Root Item Type* (page 25)
- *Test Case Action* (page 107)
- *Test Case Check* (page 107)

#### 5.2.3.79 Name

The value shall be a string. It shall be an attribute name. The value shall match with the regular expression "`^([a-z][a-z0-9-]*|SPDX-License-Identifier)$`".

This type is used by the following types:

- *Application Configuration Option Item Type* (page 41)
- *Build Item Type* (page 26)
- *External Reference* (page 79)
- *Functional Requirement Item Type* (page 49)
- *Glossary Item Type* (page 39)
- *Interface Item Type* (page 40)
- *Link* (page 89)
- *Non-Functional Requirement Item Type* (page 54)
- *Register Definition* (page 94)
- *Requirement Item Type* (page 48)
- *Requirement Validation Item Type* (page 57)
- *Root Item Type* (page 25)
- *Runtime Measurement Parameter Set* (page 98)

- *Runtime Performance Parameter Set* (page 99)
- *Specification Attribute Value* (page 100)
- *Specification Explicit Attributes* (page 100)
- *Specification Generic Attributes* (page 102)
- *Specification Item Type* (page 59)
- *Specification List* (page 104)
- *Specification Refinement Link Role* (page 105)

#### 5.2.3.80 Optional Floating-Point Number

A value of this type shall be of one of the following variants:

- The value may be a floating-point number.
- There may be no value (null).

#### 5.2.3.81 Optional Integer

A value of this type shall be of one of the following variants:

- The value may be an integer number.
- There may be no value (null).

This type is used by the following types:

- *Register Block Item Type* (page 47)

#### 5.2.3.82 Optional String

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string.

#### 5.2.3.83 Performance Runtime Limits Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `performance-runtime-limits`. It defines the performance runtime limits role of links. All explicit attributes shall be specified. The explicit attributes for this type are:

**limits**
The attribute value shall be a *Runtime Measurement Environment Table* (page 98).

#### 5.2.3.84 Placement Order Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `placement-order`. This link role defines the placement order of items in a container item (for example an interface function in a header file or a documentation section).

#### 5.2.3.85 Proxy Member Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `proxy-member`. It defines the proxy member role of links. Items may use this role to link to *Proxy Item Types* (page 48) items.

#### 5.2.3.86 Register Bits Definition

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies a register bit field. Single bits are bit fields with a width of one. All explicit attributes shall be specified. The explicit attributes for this type are:

  **brief**
  The attribute value shall be an *Interface Brief Description* (page 80).

  **description**
  The attribute value shall be an *Interface Description* (page 83).

  **name**
  The attribute value shall be a string. It shall be the name of the register bit field.

  **properties**
  The attribute value shall be a list of strings. It shall be the list of bit field properties. Properties are for example if the bit field can be read or written, or an access has side-effects such as clearing a status.

  **start**
  The attribute value shall be an integer number. It shall be the start bit of the bit field. Bit `0` is the least-significant bit.

  **width**
  The attribute value shall be an integer number. It shall be the width in bits of the bit field.

- There may be no value (null).

This type is used by the following types:

- *Register Bits Definition Directive* (page 92)
- *Register Bits Definition Variant* (page 93)

#### 5.2.3.87 Register Bits Definition Directive

This set of attributes specifies a register bits directive. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be a list. Each list element shall be a *Register Bits Definition* (page 92). The default definition will be used if no variant-specific definition is enabled.

**variants**
The attribute value shall be a list. Each list element shall be a *Register Bits Definition Variant* (page 93).

This type is used by the following types:

- *Register Definition* (page 94)

#### 5.2.3.88 Register Bits Definition Variant

This set of attributes specifies a register bits variant. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be a list. Each list element shall be a *Register Bits Definition* (page 92). The definition will be used if the expression defined by the `enabled-by` attribute evaluates to true. In generated header files, the expression is evaluated by the C preprocessor.

**enabled-by**
The attribute value shall be an *Interface Enabled-By Expression* (page 84).

This type is used by the following types:

- *Register Bits Definition Directive* (page 92)

#### 5.2.3.89 Register Block Include Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `register-block-include`. It defines the register block include role of links. Links of this role are used to build register blocks using other register blocks. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be a name to identify the included register block within the item. The name shall be unique within the scope of the item links of this role and the SpecTypeRegisterList.

#### 5.2.3.90 Register Block Member Definition

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies a register block member definition. All explicit attributes shall be specified. The explicit attributes for this type are:

  **count**
  The attribute value shall be an integer number. It shall be the count of registers of the register block member.

  **name**
  The attribute value shall be a *Register Name* (page 95).

- There may be no value (null).

This type is used by the following types:

- *Register Block Member Definition Directive* (page 93)
- *Register Block Member Definition Variant* (page 94)

#### 5.2.3.91 Register Block Member Definition Directive

This set of attributes specifies a register block member definition directive. All explicit attributes shall be specified. The explicit attributes for this type are:

**default**
The attribute value shall be a *Register Block Member Definition* (page 93). The default definition will be used if no variant-specific definition is enabled.

**offset**
The attribute value shall be an integer number. It shall be the address of the register block member relative to the base address of the register block.

**variants**
The attribute value shall be a list. Each list element shall be a *Register Block Member Definition Variant* (page 94).

This type is used by the following types:

- *Register Block Item Type* (page 47)

#### 5.2.3.92 Register Block Member Definition Variant

This set of attributes specifies a register block member definition variant. All explicit attributes shall be specified. The explicit attributes for this type are:

**definition**
The attribute value shall be a *Register Block Member Definition* (page 93). The definition will be used if the expression defined by the `enabled-by` attribute evaluates to true. In generated header files, the expression is evaluated by the C preprocessor.

**enabled-by**
The attribute value shall be an *Interface Enabled-By Expression* (page 84).

This type is used by the following types:

- *Register Block Member Definition Directive* (page 93)

#### 5.2.3.93 Register Definition

This set of attributes specifies a register. All explicit attributes shall be specified. The explicit attributes for this type are:

**bits**
The attribute value shall be a list. Each list element shall be a *Register Bits Definition Directive* (page 92).

**brief**
The attribute value shall be an *Interface Brief Description* (page 80).

**description**
The attribute value shall be an *Interface Description* (page 83).

**name**
The attribute value shall be a string. It shall be the name to identify the register definition. The name shall be unique within the scope of the *Register Block Include Role* (page 93) links of the item and the SpecTypeRegisterList.

**width**
The attribute value shall be an integer number. It shall be the width of the register in bits.

In addition to the explicit attributes, generic attributes may be specified. Each generic attribute key shall be a *Name* (page 90). The attribute value may have any type.

This type is used by the following types:

- *Register Block Item Type* (page 47)

#### 5.2.3.94 Register Name

The value shall be a string. The name consists either of an identifier, or an identifier and an alias. The identifier and alias are separated by a colon (`:`). The identifier shall match with the name of a register definition of the item (see *Register Definition* (page 94)) or the name of a register block include of the item (see *Register Block Include Role* (page 93)). If no alias is specified, then the identifier is used for the register block member name, otherwise the alias is used. If the register block member names are not unique within the item, then a postfix number is appended to the names. The number starts with zero for each set of names. The value shall match with the regular expression "`^[a-zA-Z_][a-zA-Z0-9_]*(:[a-zA-Z_][a-zA-Z0-9_]*)?$`".

This type is used by the following types:

- *Register Block Member Definition* (page 93)

#### 5.2.3.95 Requirement Design Group Identifier

A value of this type shall be of one of the following variants:

- There may be no value (null).
- The value may be a string. It shall be the identifier of the requirement design group. The value shall match with the regular expression "`^[a-zA-Z0-9_]*$`".

This type is used by the following types:

- *Design Group Requirement Item Type* (page 54)

#### 5.2.3.96 Requirement Refinement Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `requirement-refinement`. It defines the requirement refinement role of links.

#### 5.2.3.97 Requirement Text

The value shall be a string. It shall state a requirement or constraint. The text should not use one of the following words or phrases:

- acceptable
- adequate
- almost always
- and/or
- appropriate
- approximately
- as far as possible
- as much as practicable
- best
- best possible
- easy
- efficient

- e.g.
- enable
- enough
- etc.
- few
- first rate
- flexible
- generally
- goal
- graceful
- great
- greatest
- ideally
- i.e.
- if possible
- in most cases
- large
- many
- maximize
- minimize
- most
- multiple
- necessary
- numerous
- optimize
- ought to
- probably
- quick
- rapid
- reasonably
- relevant
- robust
- satisfactory
- several

- shall be included but not limited to
- simple
- small
- some
- state of the art
- sufficient
- suitable
- support
- systematically
- transparent
- typical
- user friendly
- usually
- versatile
- when necessary

This type is used by the following types:

- *Action Requirement State* (page 65)
- *Application Configuration Group Item Type* (page 41)
- *Constraint Item Type* (page 39)
- *Interface Group Item Type* (page 45)
- *Requirement Item Type* (page 48)

#### 5.2.3.98 Requirement Validation Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `validation`. It defines the requirement validation role of links.

#### 5.2.3.99 Runtime Measurement Environment Name

The value shall be a string. It specifies the runtime measurement environment name. The value

- shall be an element of
  - "`FullCache`",
  - "`HotCache`", and
  - "`DirtyCache`",
- or, shall match with the regular expression "`^Load/[1-9][0-9]*$`".

This type is used by the following types:

- *Runtime Measurement Environment Table* (page 98)

#### 5.2.3.100 Runtime Measurement Environment Table

This set of attributes provides runtime performance limits for a set of runtime measurement environments. Generic attributes may be specified. Each generic attribute key shall be a *Runtime Measurement Environment Name* (page 97). Each generic attribute value shall be a *Runtime Measurement Value Table* (page 99).

This type is used by the following types:

- *Performance Runtime Limits Link Role* (page 91)

#### 5.2.3.101 Runtime Measurement Parameter Set

This set of attributes defines parameters of the runtime measurement test case. All explicit attributes shall be specified. The explicit attributes for this type are:

**sample-count**
: The attribute value shall be an integer number. It shall be the sample count of the runtime measurement context.

In addition to the explicit attributes, generic attributes may be specified. Each generic attribute key shall be a *Name* (page 90). The attribute value may have any type.

This type is used by the following types:

- *Runtime Measurement Test Item Type* (page 58)

#### 5.2.3.102 Runtime Measurement Request Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `runtime-measurement-request`. It defines the runtime measurement request role of links. The link target shall be a *Runtime Measurement Test Item Type* (page 58) item.

#### 5.2.3.103 Runtime Measurement Value Kind

The value shall be a string. It specifies the kind of a runtime measurement value. The value shall be an element of

- “`max-lower-bound`”,
- “`max-upper-bound`”,
- “`mean-lower-bound`”,
- “`mean-upper-bound`”,
- “`median-lower-bound`”,
- “`median-upper-bound`”,
- “`min-lower-bound`”, and
- “`min-upper-bound`”.

This type is used by the following types:

- *Runtime Measurement Value Table* (page 99)

#### 5.2.3.104 Runtime Measurement Value Table

This set of attributes provides a set of runtime measurement values each of a specified kind. The unit of the values shall be one second. Generic attributes may be specified. Each generic attribute key shall be a *Runtime Measurement Value Kind* (page 98). Each generic attribute value shall be a floating-point number.

This type is used by the following types:

- *Runtime Measurement Environment Table* (page 98)

#### 5.2.3.105 Runtime Performance Parameter Set

This set of attributes defines parameters of the runtime performance requirement. Generic attributes may be specified. Each generic attribute key shall be a *Name* (page 90). The attribute value may have any type.

This type is used by the following types:

- *Runtime Performance Requirement Item Type* (page 56)

#### 5.2.3.106 SHA256 Hash Value

The value shall be a string. It shall be a SHA256 hash value encoded in base64url. The value shall match with the regular expression "`^[A-Za-z0-9+_=-]{44}$`".

This type is used by the following types:

- *External File Reference* (page 79)

#### 5.2.3.107 SPDX License Identifier

The value shall be a string. It defines the license of the item expressed though an SPDX License Identifier. The value

- shall be equal to "`CC-BY-SA-4.0 OR BSD-2-Clause`",
- or, shall be equal to "`BSD-2-Clause`",
- or, shall be equal to "`CC-BY-SA-4.0`".

This type is used by the following types:

- *Root Item Type* (page 25)

#### 5.2.3.108 Specification Attribute Set

This set of attributes specifies a set of attributes. The following explicit attributes are mandatory:

- `attributes`
- `description`
- `mandatory-attributes`

The explicit attributes for this type are:

**attributes**
The attribute value shall be a *Specification Explicit Attributes* (page 100). It shall specify the explicit attributes of the attribute set.

**description**
The attribute value shall be an optional string. It shall be the description of the attribute set.

**generic-attributes**
The attribute value shall be a *Specification Generic Attributes* (page 102). It shall specify the generic attributes of the attribute set.

**mandatory-attributes**
The attribute value shall be a *Specification Mandatory Attributes* (page 105). It shall specify the mandatory attributes of the attribute set.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.109 Specification Attribute Value

This set of attributes specifies an attribute value. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be an optional string. It shall be the description of the attribute value.

**spec-type**
The attribute value shall be a *Name* (page 90). It shall be the specification type of the attribute value.

This type is used by the following types:

- *Specification Explicit Attributes* (page 100)

#### 5.2.3.110 Specification Boolean Value

This attribute set specifies a boolean value. Only the `description` attribute is mandatory. The explicit attributes for this type are:

**assert**
The attribute value shall be a boolean. This optional attribute defines the value constraint of the specified boolean value. If the value of the assert attribute is true, then the value of the specified boolean value shall be true. If the value of the assert attribute is false, then the value of the specified boolean value shall be false. In case the assert attribute is not present, then the value of the specified boolean value may be true or false.

**description**
The attribute value shall be an optional string. It shall be the description of the specified boolean value.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.111 Specification Explicit Attributes

Generic attributes may be specified. Each generic attribute key shall be a *Name* (page 90). Each generic attribute value shall be a *Specification Attribute Value* (page 100). Each generic attribute specifies an explicit attribute of the attribute set. The key of the each generic attribute defines the attribute key of the explicit attribute.

This type is used by the following types:

- *Specification Attribute Set* (page 99)

#### 5.2.3.112 Specification Floating-Point Assert

A value of this type shall be an expression which asserts that the floating-point value of the specified attribute satisfies the required constraints.

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be a *Specification Floating-Point Assert* (page 101). The *and* operator evaluates to the *logical and* of the evaluation results of the expressions in the list.

  **eq**
  The attribute value shall be a floating-point number. The *eq* operator evaluates to true, if the value to check is equal to the value of this attribute, otherwise to false.

  **ge**
  The attribute value shall be a floating-point number. The *ge* operator evaluates to true, if the value to check is greater than or equal to the value of this attribute, otherwise to false.

  **gt**
  The attribute value shall be a floating-point number. The *gt* operator evaluates to true, if the value to check is greater than the value of this attribute, otherwise to false.

  **le**
  The attribute value shall be a floating-point number. The *le* operator evaluates to true, if the value to check is less than or equal to the value of this attribute, otherwise to false.

  **lt**
  The attribute value shall be a floating-point number. The *lt* operator evaluates to true, if the value to check is less than the value of this attribute, otherwise to false.

  **ne**
  The attribute value shall be a floating-point number. The *ne* operator evaluates to true, if the value to check is not equal to the value of this attribute, otherwise to false.

  **not**
  The attribute value shall be a *Specification Floating-Point Assert* (page 101). The *not* operator evaluates to the *logical not* of the evaluation results of the expression.

  **or**
  The attribute value shall be a list. Each list element shall be a *Specification Floating-Point Assert* (page 101). The *or* operator evaluates to the *logical or* of the evaluation results of the expressions in the list.

- The value may be a list. Each list element shall be a *Specification Floating-Point Assert* (page 101). This list of expressions evaluates to the *logical or* of the evaluation results of the expressions in the list.

This type is used by the following types:

- *Specification Floating-Point Assert* (page 101)
- *Specification Floating-Point Value* (page 102)

#### 5.2.3.113 Specification Floating-Point Value

This set of attributes specifies a floating-point value. Only the `description` attribute is mandatory. The explicit attributes for this type are:

**assert**
The attribute value shall be a *Specification Floating-Point Assert* (page 101). This optional attribute defines the value constraints of the specified floating-point value. In case the assert attribute is not present, then the value of the specified floating-point value may be every valid floating-point number.

**description**
The attribute value shall be an optional string. It shall be the description of the specified floating-point value.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.114 Specification Generic Attributes

This set of attributes specifies generic attributes. Generic attributes are attributes which are not explicitly specified by *Specification Explicit Attributes* (page 100). They are restricted to uniform attribute key and value types. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be an optional string. It shall be the description of the generic attributes.

**key-spec-type**
The attribute value shall be a *Name* (page 90). It shall be the specification type of the generic attribute keys.

**value-spec-type**
The attribute value shall be a *Name* (page 90). It shall be the specification type of the generic attribute values.

This type is used by the following types:

- *Specification Attribute Set* (page 99)

#### 5.2.3.115 Specification Information

This set of attributes specifies attribute values. At least one of the explicit attributes shall be specified. The explicit attributes for this type are:

**bool**
The attribute value shall be a *Specification Boolean Value* (page 100). It shall specify a boolean value.

**dict**
The attribute value shall be a *Specification Attribute Set* (page 99). It shall specify a set of attributes.

**float**
The attribute value shall be a *Specification Floating-Point Value* (page 102). It shall specify a floating-point value.

**int**
The attribute value shall be a *Specification Integer Value* (page 104). It shall specify an integer value.

**list**
The attribute value shall be a *Specification List* (page 104). It shall specify a list of attributes or values.

**none**
The attribute shall have no value. It specifies that no value is required.

**str**
The attribute value shall be a *Specification String Value* (page 107). It shall specify a string.

This type is used by the following types:

- *Specification Item Type* (page 59)

#### 5.2.3.116 Specification Integer Assert

A value of this type shall be an expression which asserts that the integer value of the specified attribute satisfies the required constraints.

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be a *Specification Integer Assert* (page 103). The *and* operator evaluates to the *logical and* of the evaluation results of the expressions in the list.

  **eq**
  The attribute value shall be an integer number. The *eq* operator evaluates to true, if the value to check is equal to the value of this attribute, otherwise to false.

  **ge**
  The attribute value shall be an integer number. The *ge* operator evaluates to true, if the value to check is greater than or equal to the value of this attribute, otherwise to false.

  **gt**
  The attribute value shall be an integer number. The *gt* operator evaluates to true, if the value to check is greater than the value of this attribute, otherwise to false.

  **le**
  The attribute value shall be an integer number. The *le* operator evaluates to true, if the value to check is less than or equal to the value of this attribute, otherwise to false.

  **lt**
  The attribute value shall be an integer number. The *lt* operator evaluates to true, if the value to check is less than the value of this attribute, otherwise to false.

**ne**
The attribute value shall be an integer number. The *ne* operator evaluates to true, if the value to check is not equal to the value of this attribute, otherwise to false.

**not**
The attribute value shall be a *Specification Integer Assert* (page 103). The *not* operator evaluates to the *logical not* of the evaluation results of the expression.

**or**
The attribute value shall be a list. Each list element shall be a *Specification Integer Assert* (page 103). The *or* operator evaluates to the *logical or* of the evaluation results of the expressions in the list.

- The value may be a list. Each list element shall be a *Specification Integer Assert* (page 103). This list of expressions evaluates to the *logical or* of the evaluation results of the expressions in the list.

This type is used by the following types:

- *Specification Integer Assert* (page 103)
- *Specification Integer Value* (page 104)

#### 5.2.3.117 Specification Integer Value

This set of attributes specifies an integer value. Only the `description` attribute is mandatory. The explicit attributes for this type are:

**assert**
The attribute value shall be a *Specification Integer Assert* (page 103). This optional attribute defines the value constraints of the specified integer value. In case the assert attribute is not present, then the value of the specified integer value may be every valid integer number.

**description**
The attribute value shall be an optional string. It shall be the description of the specified integer value.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.118 Specification List

This set of attributes specifies a list of attributes or values. All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be an optional string. It shall be the description of the list.

**spec-type**
The attribute value shall be a *Name* (page 90). It shall be the specification type of elements of the list.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.119 Specification Mandatory Attributes

It defines which explicit attributes are mandatory.

A value of this type shall be of one of the following variants:

- The value may be a list. Each list element shall be a *Name* (page 90). The list defines the mandatory attributes through their key names.
- The value may be a string. It defines how many explicit attributes are mandatory. If `none` is used, then none of the explicit attributes is mandatory, they are all optional. The value shall be an element of
  - "`all`",
  - "`at-least-one`",
  - "`at-most-one`",
  - "`exactly-one`", and
  - "`none`".

This type is used by the following types:

- *Specification Attribute Set* (page 99)

#### 5.2.3.120 Specification Member Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `spec-member`. It defines the specification membership role of links.

#### 5.2.3.121 Specification Refinement Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `spec-refinement`. It defines the specification refinement role of links. All explicit attributes shall be specified. The explicit attributes for this type are:

**spec-key**
The attribute value shall be a *Name* (page 90). It shall be the specification type refinement attribute key of the specification refinement.

**spec-value**
The attribute value shall be a *Name* (page 90). It shall be the specification type refinement attribute value of the specification refinement.

#### 5.2.3.122 Specification String Assert

A value of this type shall be an expression which asserts that the string of the specified attribute satisfies the required constraints.

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. Each attribute defines an operator. Exactly one of the explicit attributes shall be specified. The explicit attributes for this type are:

  **and**
  The attribute value shall be a list. Each list element shall be a *Specification String Assert* (page 105). The *and* operator evaluates to the *logical and* of the evaluation results of the expressions in the list.

**contains**
The attribute value shall be a list of strings. The *contains* operator evaluates to true, if the string to check converted to lower case with all white space characters converted to a single space character contains a string of the list of strings of this attribute, otherwise to false.

**eq**
The attribute value shall be a string. The *eq* operator evaluates to true, if the string to check is equal to the value of this attribute, otherwise to false.

**ge**
The attribute value shall be a string. The *ge* operator evaluates to true, if the string to check is greater than or equal to the value of this attribute, otherwise to false.

**gt**
The attribute value shall be a string. The *gt* operator evaluates to true, if the string to check is greater than the value of this attribute, otherwise to false.

**in**
The attribute value shall be a list of strings. The *in* operator evaluates to true, if the string to check is contained in the list of strings of this attribute, otherwise to false.

**le**
The attribute value shall be a string. The *le* operator evaluates to true, if the string to check is less than or equal to the value of this attribute, otherwise to false.

**lt**
The attribute value shall be a string. The *lt* operator evaluates to true, if the string to check is less than the value of this attribute, otherwise to false.

**ne**
The attribute value shall be a string. The *ne* operator evaluates to true, if the string to check is not equal to the value of this attribute, otherwise to false.

**not**
The attribute value shall be a *Specification String Assert* (page 105). The *not* operator evaluates to the *logical not* of the evaluation results of the expression.

**or**
The attribute value shall be a list. Each list element shall be a *Specification String Assert* (page 105). The *or* operator evaluates to the *logical or* of the evaluation results of the expressions in the list.

**re**
The attribute value shall be a string. The *re* operator evaluates to true, if the string to check matches with the regular expression of this attribute, otherwise to false.

**uid**
The attribute shall have no value. The *uid* operator evaluates to true, if the string is a valid UID, otherwise to false.

- The value may be a list. Each list element shall be a *Specification String Assert* (page 105). This list of expressions evaluates to the *logical or* of the evaluation results of the expressions in the list.

This type is used by the following types:

- *Specification String Assert* (page 105)

- *Specification String Value* (page 107)

#### 5.2.3.123 Specification String Value

This set of attributes specifies a string. Only the `description` attribute is mandatory. The explicit attributes for this type are:

**assert**
The attribute value shall be a *Specification String Assert* (page 105). This optional attribute defines the constraints of the specified string. In case the assert attribute is not present, then the specified string may be every valid string.

**description**
The attribute value shall be an optional string. It shall be the description of the specified string attribute.

This type is used by the following types:

- *Specification Information* (page 102)

#### 5.2.3.124 Test Case Action

This set of attributes specifies a test case action. All explicit attributes shall be specified. The explicit attributes for this type are:

**action-brief**
The attribute value shall be an optional string. It shall be the test case action brief description.

**action-code**
The attribute value shall be a string. It shall be the test case action code.

**checks**
The attribute value shall be a list. Each list element shall be a *Test Case Check* (page 107).

**links**
The attribute value shall be a list. Each list element shall be a *Link* (page 89). The links should use the *Requirement Validation Link Role* (page 97) for validation tests and the *Unit Test Link Role* (page 110) for unit tests.

This type is used by the following types:

- *Test Case Item Type* (page 60)

#### 5.2.3.125 Test Case Check

This set of attributes specifies a test case check. All explicit attributes shall be specified. The explicit attributes for this type are:

**brief**
The attribute value shall be an optional string. It shall be the test case check brief description.

**code**
The attribute value shall be a string. It shall be the test case check code.

**links**
The attribute value shall be a list. Each list element shall be a *Link* (page 89). The links should use the *Requirement Validation Link Role* (page 97) for validation tests and the *Unit Test Link Role* (page 110) for unit tests.

This type is used by the following types:

- *Test Case Action* (page 107)

#### 5.2.3.126 Test Context Member

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes defines an action requirement test context member. All explicit attributes shall be specified. The explicit attributes for this type are:

  **brief**
  The attribute value shall be an optional string. It shall be the test context member brief description.

  **description**
  The attribute value shall be an optional string. It shall be the test context member description.

  **member**
  The attribute value shall be a string. It shall be the test context member definition. It shall be a valid C structure member definition without a trailing `;`.

- There may be no value (null).

This type is used by the following types:

- *Action Requirement Item Type* (page 49)
- *Runtime Measurement Test Item Type* (page 58)
- *Test Case Item Type* (page 60)

#### 5.2.3.127 Test Header

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes specifies a test header. In case a test header is specified, then instead of a test case a test run function will be generated. The test run function will be declared in the test header target file and defined in the test source target file. The test run function can be used to compose test cases. The test header file is not automatically included in the test source file. It should be added to the includes or local includes of the test. All explicit attributes shall be specified. The explicit attributes for this type are:

  **code**
  The attribute value shall be an optional string. If the value is present, then it shall be the test header code. The header code is placed at file scope after the general test declarations and before the test run function declaration.

  **freestanding**
  The attribute value shall be a boolean. The value shall be `true`, if the test case is free-standing, otherwise `false`. Freestanding test cases are not statically registered. Instead the generated test runner uses `T_case_begin()` and `T_case_end()`.

  **includes**
  The attribute value shall be a list of strings. It shall be a list of header files included by the header file via `#include <...>`.

**local-includes**
The attribute value shall be a list of strings. It shall be a list of header files included by the header file via `#include "..."`.

**run-params**
The attribute value shall be a list. Each list element shall be a *Test Run Parameter* (page 109).

**target**
The attribute value shall be a string. It shall be the path to the generated test header file.

- There may be no value (null).

This type is used by the following types:

- *Action Requirement Item Type* (page 49)
- *Test Case Item Type* (page 60)

#### 5.2.3.128 Test Run Parameter

This set of attributes specifies a parameter for the test run function. In case this parameter is used in an *Action Requirement Item Type* (page 49) item, then the parameter is also added as a member to the test context, see *Test Context Member* (page 108). All explicit attributes shall be specified. The explicit attributes for this type are:

**description**
The attribute value shall be a string. It shall be the description of the parameter.

**dir**
The attribute value shall be an *Interface Parameter Direction* (page 88).

**name**
The attribute value shall be a string. It shall be the parameter name.

**specifier**
The attribute value shall be a string. It shall be the complete function parameter specifier. Use `${.:name}` for the parameter name, for example `"int ${.:name}"`.

This type is used by the following types:

- *Test Header* (page 108)

#### 5.2.3.129 Test Support Method

A value of this type shall be of one of the following variants:

- The value may be a set of attributes. This set of attributes defines an action requirement test support method. All explicit attributes shall be specified. The explicit attributes for this type are:

  **brief**
  The attribute value shall be an optional string. It shall be the test support method brief description.

  **code**
  The attribute value shall be a string. It shall be the test support method code. The code may use a local variable `ctx` which points to the test context, see *Test Context Member* (page 108).

**description**
The attribute value shall be an optional string. It shall be the test support method description.

- There may be no value (null).

This type is used by the following types:

- *Action Requirement Item Type* (page 49)
- *Runtime Measurement Test Item Type* (page 58)
- *Runtime Performance Requirement Item Type* (page 56)
- *Test Case Item Type* (page 60)

#### 5.2.3.130 UID

The value shall be a string. It shall be a valid absolute or relative item UID.

This type is used by the following types:

- *Link* (page 89)

#### 5.2.3.131 Unit Test Link Role

This type refines the *Link* (page 89) through the `role` attribute if the value is `unit-test`. It defines the unit test role of links. For unit tests the link target should be the *Interface Domain Item Type* (page 43) containing the software unit. All explicit attributes shall be specified. The explicit attributes for this type are:

**name**
The attribute value shall be a string. It shall be the name of the tested software unit.

## 5.3 Traceability of Specification Items

The standard ECSS-E-ST-10-06C demands that requirements shall be under configuration management, backwards-traceable and forward-traceable [ECS09]. Requirements are a specialization of specification items in RTEMS.

### 5.3.1 History of Specification Items

The RTEMS specification items should placed in the RTEMS sources using Git for version control. The history of specification items can be traced with Git. Special commit procedures for changes in specification item files should be established. For example, it should be allowed to change only one specification item per commit. A dedicated Git commit message format may be used as well, e.g. use of `Approved-by:` or `Reviewed-by:` lines which indicate an agreed statement (similar to the Linux kernel patch submission guidelines). Git commit procedures may be ensured through a server-side pre-receive hook. The history of requirements may be also added to the specification items directly in a *revision* attribute. This would make it possible to generate the history information for documents without having the Git repository available, e.g. from an RTEMS source release archive.

### 5.3.2 Backward Traceability of Specification Items

Providing backward traceability of specification items means that we must be able to find the corresponding higher level specification item for each refined specification item. A custom tool needs to verify this.

### 5.3.3 Forward Traceability of Specification Items

Providing forward traceability of specification items means that we must be able to find all the refined specification items for each higher level specification item. A custom tool needs to verify this. The links from parent to child specification items are implicitly defined by links from a child item to a parent item.

### 5.3.4 Traceability between Software Requirements, Architecture and Design

The software requirements are implemented in custom YAML files, see *Specification Items* (page 24). The software architecture and design is written in Doxygen markup. Doxygen markup is used throughout all header and source files. A Doxygen filter program may be provided to place Doxygen markup in assembler files. The software architecture is documented via Doxygen groups. Each Doxygen group name should have a project-specific name and the name should be unique within the project, e.g. RTEMSTopLevelMidLevelLowLevel. The link from a Doxygen group to its parent group is realized through the `@ingroup` special command. The link from a Doxygen group or *software component* to the corresponding requirement is realized through a `@satisfy{req}` custom command which needs the identifier of the requirement as its one and only parameter. Only links to parents are explicitly given in the Doxygen markup. The links from a parent to its children are only implicitly specified via the link from a child to its parent. So, a tool must process all files to get the complete hierarchy of software requirements, architecture and design. Links from a software component to another software component are realized through automatic Doxygen references or the `@ref` and `@see` special commands.

## 5.4 Requirement Management

### 5.4.1 Change Control Board

Working with requirements usually involves a Change Control Board (*CCB*). The CCB of the RTEMS Project is the RTEMS developer mailing list.

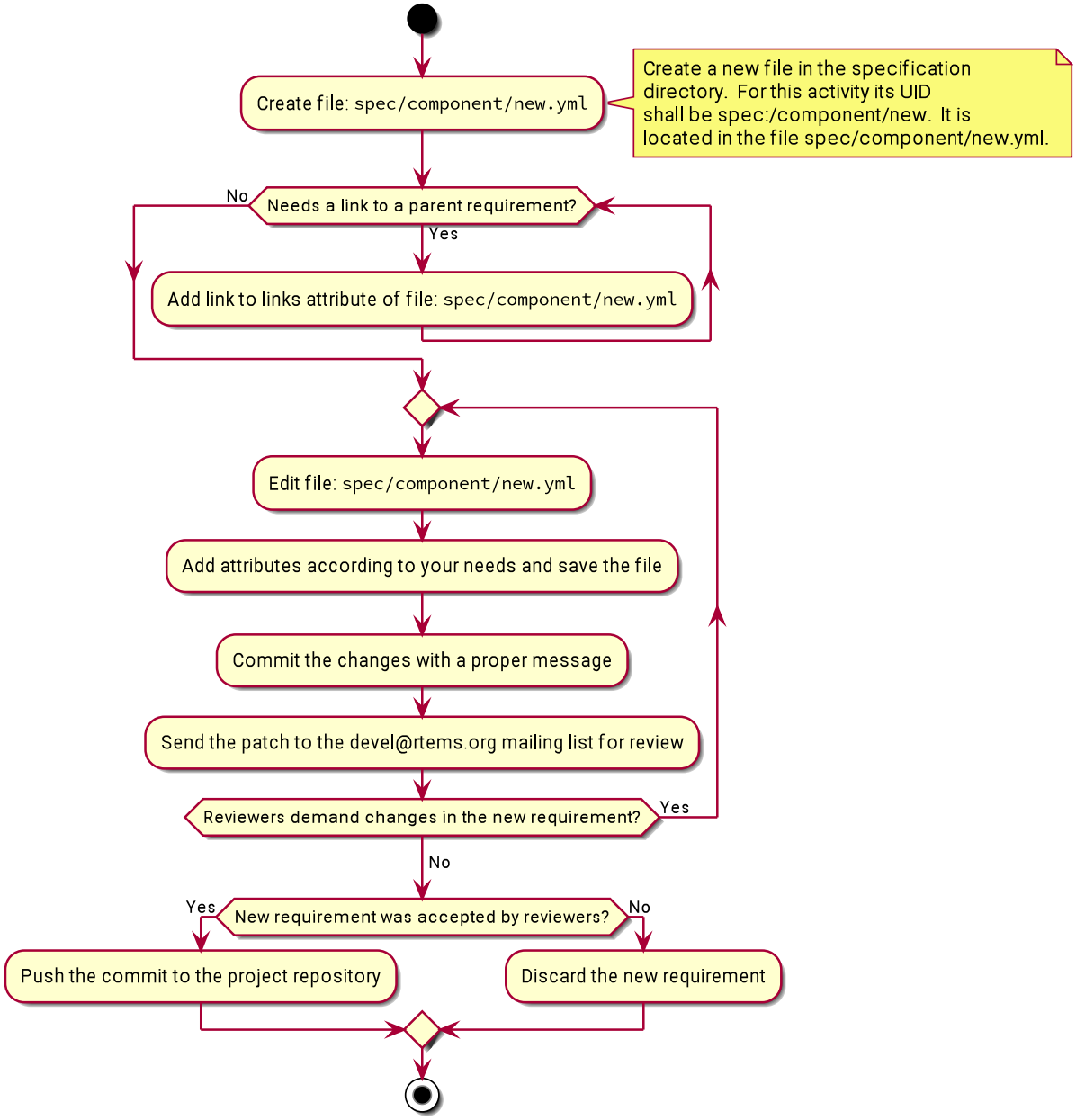
There are the following actors involved:

- *RTEMS users*: Everyone using the RTEMS real-time operating system to design, develop and build an application on top of it.
- *RTEMS developers*: The persons developing and maintaining RTEMS. They write patches to add or modify code, requirements, tests and documentation.

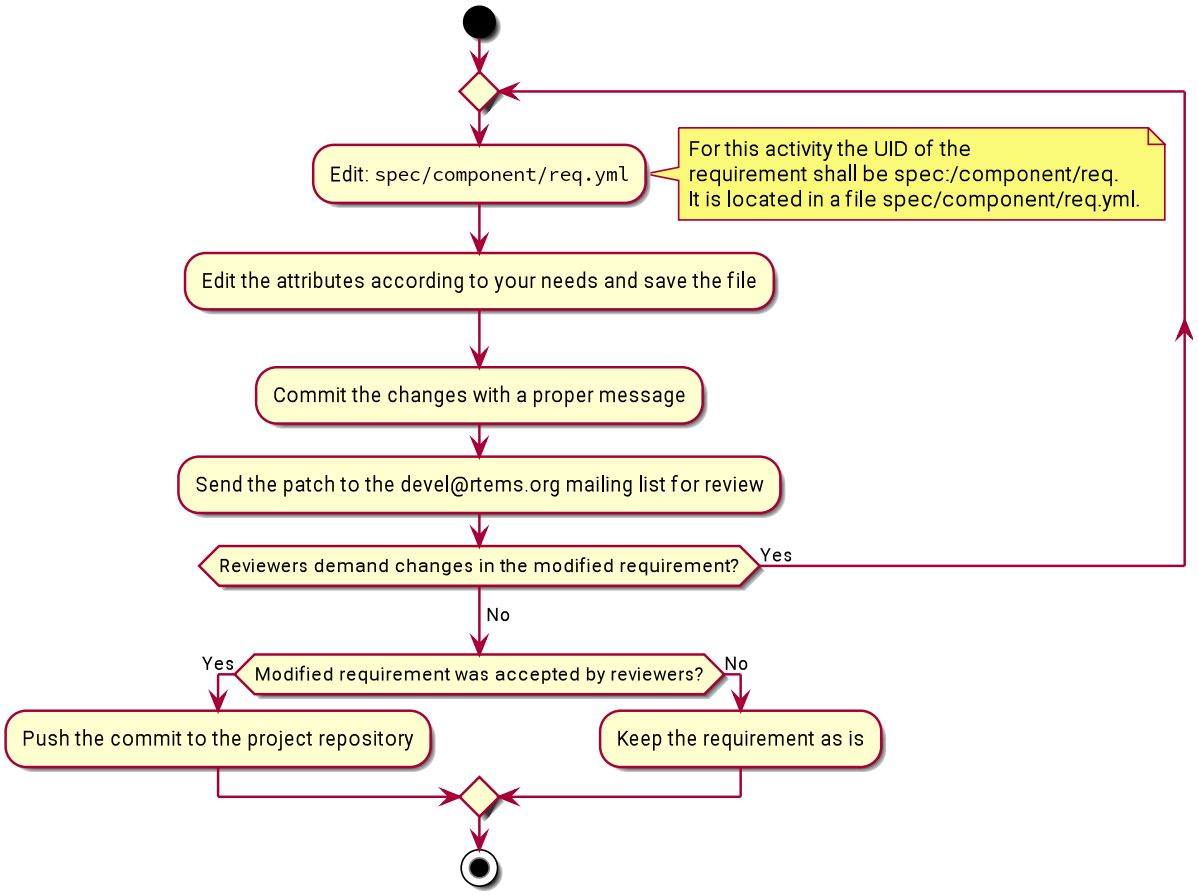
Adding and changing requirements follows the normal patch review process. The normal patch review process is described in the RTEMS User Manual. Reviews and comments may be submitted by anyone, but a maintainer review is required to approve *significant* changes. In addition for significant changes, there should be at least one reviewer with a sufficient independence from the author which proposes a new requirement or a change of an existing requirement. Working in another company on different projects is sufficiently independent. RTEMS maintainers do not know all the details, so they trust in general people with experience on a certain platform. Sometimes no review comments may appear in a reasonable time frame, then an implicit agreement to the proposed changes is assumed. Patches can be sent at anytime, so controlling changes in RTEMS requires a permanent involvement on the RTEMS developer mailing list.

For a qualification of RTEMS according to certain standards, the requirements may be approved by an RTEMS user. The approval by RTEMS users is not the concern of the RTEMS Project, however, the RTEMS Project should enable RTEMS users to manage the approval of requirements easily. This information may be also used by a independent authority which comes into play with an Independent Software Verification and Validation (*ISVV*). It could be used to select a subset of requirements, e.g. look only at the ones approved by a certain user. RTEMS users should be able to reference the determinative content of requirements, test procedures, test cases and justification reports in their own documentation. Changes in the determinative content should invalidate all references to previous versions.

### 5.4.2 Add a Requirement



### 5.4.3 Modify a Requirement




### 5.4.4 Mark a Requirement as Obsolete

Requirements shall be never removed. They shall be marked as obsolete. This ensures that requirement identifiers are not reused. The procedure to obsolete a requirement is the same as the one to *modify a requirement* (page 114).

## 5.5 Tooling

### 5.5.1 Tool Requirements

To manage requirements some tool support is helpful. Here is a list of requirements for the tool:

- The tool shall be open source.
- The tool should be actively maintained during the initial phase of the RTEMS requirements specification.
- The tool shall use plain text storage (no binary formats, no database).
- The tool shall support version control via Git.
- The tool should export the requirements in a human readable form using the Sphinx documentation framework.
- The tool shall support traceability of requirements to items external to the tool.
- The tool shall support traceability between requirements.
- The tool shall support custom requirement attributes.
- The tool should ensure that there are no cyclic dependencies between requirements.
- The tool should provide an export to *ReqIF*.

### 5.5.2 Tool Evaluation

During an evaluation phase the following tools were considered:

- aNimble
- *Doorstop*
- OSRMT
- Papyrus
- ProR
- ReqIF Studio
- Requirement Heap
- rmToo

The tools aNimble, OSRMT and Requirement Heap were not selected since they use a database. The tools Papyrus, ProR and ReqIF are Eclipse based and use complex XML files for data storage. They were difficult to use and lack good documentation/tutorials. The tools rmToo and Doorstop turned out to be the best candidates to manage requirements in the RTEMS Project. The Doorstop tool was selected as the first candidate mainly due a recommendation by an RTEMS user.

### 5.5.3 Best Available Tool - Doorstop

*Doorstop* is a requirements management tool. It has a modern, object-oriented and well-structured implementation in Python 3.6 under the LGPLv3 license. It uses a continuous integration build with style checkers, static analysis, documentation checks, code coverage, unit test and integration tests. In 2019, the project was actively maintained. Pull requests for minor improvements and new features were reviewed and integrated within days. Each requirement

is contained in a single file in *YAML* format. Requirements are organized in documents and can be linked to each other [BA14].

Doorstop consists of three main parts

- a stateless command line tool `doorstop`,
- a file format with a pre-defined set of attributes (YAML), and
- a primitive GUI tool (not intended to be used).

For RTEMS, its scope could be extended to manage specifications in general. The primary reason for a close consideration of Doorstop as the requirements management tool for the RTEMS Project was its data format which allows a high degree of customization. Doorstop uses a directed, acyclic graph (DAG) of items. The items are files in YAML format. Each item has a set of standard attributes (key-value pairs).

The use case for the standard attributes is requirements management. However, Doorstop is capable to manage custom attributes as well. We will heavily use custom attributes for the specification items. Enabling Doorstop to effectively use custom attributes was done specifically for the RTEMS Project in several patch sets which in the end turned out to be not enough to use Doorstop for the RTEMS Project.

A key feature of Doorstop is the fingerprint of items. For the RTEMS Project, the fingerprint hash algorithm was changed from MD5 to SHA256. In 2019, it can be considered cryptographically secure. The fingerprint should cover the normative values of an item, e.g. comments etc. are not included. The fingerprint would help RTEMS users to track the significant changes in the requirements (in contrast to all the changes visible in Git). As an example use case, a user may want to assign a project-specific status to specification items. This can be done with a table which contains columns for

1. the UID of the item,
2. the fingerprint, and
3. the project-specific status.

Given the source code of RTEMS (which includes the specification items) and this table, it can be determined which items are unchanged and which have another status (e.g. unknown, changed, etc.).

After some initial work with Doorstop some issues surfaced (#471). It turned out that Doorstop is not designed as a library and contains too much policy. This results in a lack of flexibility required for the RTEMS Project.

1. Its primary use case is requirements management. So, it has some standard attributes useful in this domain, like derived, header, level, normative, ref, reviewed, and text. However, we want to use it more generally for specification items and these attributes make not always sense. Having them in every item is just overhead and may cause confusion.
2. The links cannot have custom attributes, e.g. role, enabled-by. With link-specific attributes you could have multiple DAGs formed up by the same set of items.
3. Inside a document (directory) items are supposed to have a common type (set of attributes). We would like to store at a hierarchy level also distinct specializations.
4. The verification of the items is quite limited. We need verification with type-based rules.
5. The UIDs in combination with the document hierarchy lead to duplication, e.g. a/b/c/a-b-c-d.yml. You have the path (a/b/c) also in the file name (a-b-c). You cannot have

relative UIDs in links (e.g. .../parent-req) . The specification items may contain multiple requirements, e.g. min/max attributes. There is no way to identify them.

6. The links are ordered by Doorstop alphabetically by UID. For some applications, it would be better to use the order specified by the user. For example, we want to use specification items for a new build system. Here it is handy if you can express things like this: A is composed of B and C. Build B before C.

### 5.5.4 Custom Requirements Management Tool

No requirements management tool was available that fits the need of the RTEMS Qualification Project. The decision was to develop a custom requirements management tool written in Python 3.6 or later. The design for it is heavily inspired by Doorstop.

## 5.6 How-To

### 5.6.1 Getting Started

The RTEMS specification items and qualification tools are work in progress. The first step to work with the RTEMS specification and the corresponding tools is a clone of the following repository:

```
git clone https://gitlab.rtems.org/rtems/prequal/rtems-central.git
git submodule init
git submodule update
```

The tools need a virtual Python 3 environment. To set it up use:

```
cd rtems-central
make env
```

Each time you want to use one of the tools, you have to activate the environment in your shell:

```
cd rtems-central
. env/bin/activate
```

### 5.6.2 View the Specification Graph

The specification items form directed graphs through *Link* (page 89) attributes. Each link has a role. For a particular view only specific roles may be of interest. For example, the requirements specification of RTEMS starts with the `spec:/req/root` specification item. It should form a tree (connected graph without cycles). A text representation of the tree can be printed with the `./specview.py` script:

```
cd rtems-central
. env/bin/activate
./specview.py
```

This gives the following example output (shortened):

```
/req/root (type=requirement/non-functional/design)
  /bsp/if/group (type=requirement/non-functional/design-group, role=requirement-
↪refinement)
    /bsp/if/acfg-idle-task-body (type=interface/unspecified-define,␣
↪role=interface-ingroup)
      /bsp/sparc/leon3/req/idle-task-body (type=requirement/functional/function,␣
↪role=interface-function)
        /bsp/sparc/leon3/req/idle-task-power-down (type=requirement/functional/
↪function, role=requirement-refinement)
          /bsp/sparc/leon3/val/errata-gr712rc-08 (type=validation,␣
↪role=validation)
        /bsp/sparc/leon3/req/idle-task-power-down-errata (type=requirement/
↪functional/function, role=requirement-refinement)
          /bsp/sparc/leon3/val/errata-gr712rc-08 (type=validation,␣
↪role=validation)
```

The actual specification graph depends on build configuration options which enable or disable specification items. The `--enabled` command line option may be used to specify the build configuration options, for example `--enabled=sparc,bsps/sparc/leon3,sparc/gr740,RTEMS_SMP,RTEMS_QUAL`.

The `./specview.py` script can display other views of the specification through the `--filter` command line option. Transition maps of *Action Requirement Item Type* (page 49) items can be printed using the `--filter=action-table` or `--filter=action-list` filters. For example, `./specview.py --filter=action-table /rtems/timer/req/create` prints something like this:

```
.. table::
    :class: longtable

    ===== ========== ======= ===== ==== ======= ======= =====
    Entry Descriptor Name    Id    Free Status  Name    IdVar
    ===== ========== ======= ===== ==== ======= ======= =====
    0     0          Valid   Valid Yes  Ok      Valid   Set
    1     0          Valid   Valid No   TooMany Invalid Nop
    2     0          Valid   Null  Yes  InvAddr Invalid Nop
    3     0          Valid   Null  No   InvAddr Invalid Nop
    4     0          Invalid Valid Yes  InvName Invalid Nop
    5     0          Invalid Valid No   InvName Invalid Nop
    6     0          Invalid Null  Yes  InvName Invalid Nop
    7     0          Invalid Null  No   InvName Invalid Nop
    ===== ========== ======= ===== ==== ======= ======= =====
```

For example, `./specview.py --filter=action-list /rtems/timer/req/create` prints something like this:

```
Status = Ok, Name = Valid, IdVar = Set

    * Name = Valid, Id = Valid, Free = Yes

Status = TooMany, Name = Invalid, IdVar = Nop

    * Name = Valid, Id = Valid, Free = No

Status = InvAddr, Name = Invalid, IdVar = Nop

    * Name = Valid, Id = Null, Free = { Yes, No }

Status = InvName, Name = Invalid, IdVar = Nop

    * Name = Invalid, Id = { Valid, Null }, Free = { Yes, No }
```

The view above yields for each variation of post-condition states the list of associated pre-condition state variations.

### 5.6.3 Generate Files from Specification Items

The `./spec2modules.py` script generates program and documentation files in `modules/rtems` and `modules/rtems-docs` using the specification items as input. The script should be invoked whenever a specification item was modified. After running the script, go into the subdirectories and create corresponding patch sets. For these patch sets, the normal patch review process applies, see *Support and Contributing* chapter of the *RTEMS User Manual*.

### 5.6.4 Application Configuration Options

The application configuration options and groups are maintained by specification items in the directory `spec/acfg/if`. Application configuration options are grouped by *Application Configuration Group Item Type* (page 41) items which should be stored in files using the `spec/acfg/if/group-*.yml` pattern. Each application configuration option shall link to exactly one group item with the *Interface Group Membership Link Role* (page 87). There are four application option item types available which cover all existing options:

- The *feature enable options* let the application enable a feature option. If the option is not defined, then the feature is simply not available or active. There should be no feature-specific code linked to the application if the option is not defined. Examples are options which enable a device driver like `CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`. These options are specified by *Application Configuration Feature Enable Option Item Type* (page 42) items.
- The *feature options* let the application enable a specific feature option. If the option is not defined, then a default feature option is used. Regardless whether the option is defined or not defined, feature-specific code may be linked to the application. Examples are options which disable a feature if the option is defined such as `CONFIGURE_APPLICATION_DISABLE_FILESYSTEM` and options which provide a stub implementation of a feature by default and a full implementation if the option is defined such as `CONFIGURE_IMFS_ENABLE_MKFIFO`. These options are specified by *Application Configuration Feature Option Item Type* (page 42) items.
- The *integer value options* let the application define a specific value for a system parameter. If the option is not defined, then a default value is used for the system parameter. Examples are options which define the maximum count of objects available for application use such as `CONFIGURE_MAXIMUM_TASKS`. These options are specified by *Application Configuration Value Option Item Type* (page 42) items.
- The *initializer options* let the application define a specific initializer for a system parameter. If the option is not defined, then a default setting is used for the system parameter. An example option of this type is `CONFIGURE_INITIAL_EXTENSIONS`. These options are specified by *Application Configuration Value Option Item Type* (page 42) items.

Sphinx documentation sources and header files with Doxygen markup are generated from the specification items. The descriptions in the items shall use a restricted Sphinx formatting. Emphasis via one asterisk (“*”), strong emphasis via two asterisk (“**”), code samples via blockquotes (“``”), code blocks (“... code-block:: c”) and lists are allowed. References to interface items are also allowed, for example “${appl-needs-clock-driver:/name}” and “${/rtems/task/if/create:/name}”. References to other parts of the documentation are possible, however, they have to be provided by `spec:/doc/if/*` items.

#### 5.6.4.1 Modify an Existing Group

Search for the group by its section header and edit the specification item file. For example:

```
$ grep -rl "name: General System Configuration" spec/acfg/if
spec/acfg/if/group-general.yml
$ vi spec/acfg/if/group-general.yml
```

#### 5.6.4.2 Modify an Existing Option

Search for the option by its C preprocessor define name and edit the specification item file. For example:

```
$ grep -rl CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER spec/acfg/if
spec/acfg/if/appl-needs-clock-driver.yml
$ vi spec/acfg/if/appl-needs-clock-driver.yml
```

#### 5.6.4.3 Add a New Group

Let `new` be the UID name part of the new group. Create the file `spec/acfg/if/group-new.yml` and provide all attributes for an *Application Configuration Group Item Type* (page 41) item. For example:

```
$ vi spec/acfg/if/group-new.yml
```

#### 5.6.4.4 Add a New Option

Let `my-new-option` be the UID name of the option. Create the file `if/acfg/my-new-option.yml` and provide all attributes for an appropriate refinement of *Application Configuration Option Item Type* (page 41). For example:

```
$ vi spec/acfg/if/my-new-option.yml
```

#### 5.6.4.5 Generate Content after Changes

Once you are done with the modifications of an existing item or the creation of a new item, the changes need to be propagated to generated source files. This is done by the `spec2modules.py` script. Before you call this script, make sure the Git submodules are up-to-date.

```
$ ./spec2modules.py
```

The script modifies or creates source files in `modules/rtems` and `modules/rtems-docs`. Create patch sets for these changes just as if these were work done by a human and follow the normal patch review process described in the *RTEMS User Manual*. When the changes are integrated, update the Git submodules and check in the changed items.

### 5.6.5 Glossary Specification

The glossary of terms for the RTEMS Project is defined by *Glossary Term Item Type* (page 40) items in the `spec/glossary` directory. For a new glossary term add a glossary item to this directory. As the file name use the term in lower case with all white space and special characters removed or replaced by alphanumeric characters, for example `spec/glossary/magicpower.yml` for the term `magic power`.

Use `${uid:/attribute}` substitutions to reference other parts of the specification.

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
glossary-type: term
links:
- role: glossary-member
  uid: ../glossary-general
term: magic power
text: |
  Magic power enables a caller to create magic objects using a
  ${magicwand:/term}.
type: glossary
```

Define acronyms with the phrase `This term is an acronym for *.` in the `text` attribute:

```
...
term: MP
...
text: |
  This term is an acronym for Magic Power.
...
```

Once you are done with the glossary items, run the script `spec2modules.py` to generate the derived documentation content. Send patches for the generated documentation and the specification to the Developers Mailing List and follow the normal patch review process.

### 5.6.6 Interface Specification

#### 5.6.6.1 Specify an API Header File

The RTEMS *API* header files are specified under `spec:/rtems/*/if`. Create a subdirectory with a corresponding name for the API, for example in `spec/rtems/foo/if` for the `foo` API. In this new subdirectory place an *Interface Header File Item Type* (page 45) item named `header.yml` (`spec/rtems/foo/if/header.yml`) and populate it with the required attributes.

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
enabled-by: true
interface-type: header-file
links:
- role: interface-placement
  uid: /if/domain
- role: interface-ingroup
  uid: ../req/group
path: rtems/rtems/foo.h
prefix: cpukit/include
type: interface
```

#### 5.6.6.2 Specify an API Element

Figure out the corresponding header file item. If it does not exist, see *Specify an API Header File* (page 122). Place a specialization of an *Interface Item Type* (page 40) item into the directory of the header file item, for example `spec/rtems/foo/if/bar.yml` for the `bar()` function. Add the required attributes for the new interface item. Do not hard code interface names which are used to define the new interface. Use `${uid-of-interface-item:/name}` instead. If the referenced interface is specified in the same directory, then use a relative UID. Using interface references creates implicit dependencies and helps the header file generator to resolve the interface dependencies and header file includes for you. Use *Interface Unspecified Item Type* (page 46) items for interface dependencies to other domains such as the C language, the compiler, the implementation, or user-provided defines. To avoid cyclic dependencies between types you may use an *Interface Forward Declaration Item Type* (page 44) item.

```yaml
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
brief: Tries to create a magic object and returns it.
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
definition:
  default:
    body: null
    params:
    - ${magic-wand:/name} ${.:/params[0]/name}
    return: ${magic-type:/name} *
  variants: []
description: |
  The magic object is created out of nothing with the help of a magic wand.
enabled-by: true
interface-type: function
links:
- role: interface-placement
  uid: header
- role: interface-ingroup
  uid: /groups/api/classic/foo
name: bar
notes: null
params:
- description: is the magic wand.
  dir: null
  name: magic_wand
return:
  return: Otherwise, the magic object is returned.
  return-values:
  - description: The caller did not have enough magic power.
    value: ${/c/if/null}
type: interface
```

### 5.6.7 Requirements Depending on Build Configuration Options

Use the `enabled-by` attribute of items or parts of an item to make it dependent on build configuration options such as `RTEMS_SMP` or architecture-specific options such as `CPU_ENABLE_ROBUST_THREAD_DISPATCH`, see *Enabled-By Expression* (page 77). With this attribute the specification can be customized at the level of an item or parts of an item. If the `enabled-by` attribute evaluates to false for a particular configuration, then the item or the associated part is disabled in the specification. The `enabled-by` attribute acts as a formalized *where* clause, see *recommended requirements syntax* (page 19).

Please have a look at the following example which specifies the transition map of `rtems_signal_catch()`:

```
transition-map:
- enabled-by: true
  post-conditions:
    Status: Ok
    ASRInfo:
    - if:
        pre-conditions:
          Handler: Valid
      then: New
    - else: Inactive
  pre-conditions:
    Pending: all
    Handler: all
    Preempt: all
    Timeslice: all
    ASR: all
    IntLvl: all
- enabled-by: CPU_ENABLE_ROBUST_THREAD_DISPATCH
  post-conditions:
    Status: NotImplIntLvl
    ASRInfo: NopIntLvl
  pre-conditions:
    Pending: all
    Handler:
    - Valid
    Preempt: all
    Timeslice: all
    ASR: all
    IntLvl:
    - Positive
- enabled-by: RTEMS_SMP
  post-conditions:
    Status: NotImplNoPreempt
    ASRInfo: NopNoPreempt
  pre-conditions:
    Pending: all
    Handler:
    - Valid
    Preempt:
```

(continues on next page)

(continued from previous page)

```
    - 'No'
    Timeslice: all
    ASR: all
    IntLvl: all
```

### 5.6.8 Requirements Depending on Application Configuration Options

Requirements which depend on application configuration options such as `CONFIGURE_MAXIMUM_PROCESSORS` should be written in the following *syntax* (page 19):

> **Where** <feature is included>, the <system name> shall <system response>.

Use these clauses with care. Make sure all feature combinations are covered. Using a truth table may help. If a requirement depends on multiple features, use:

> **Where** <feature 0>, **where** <feature 1>, **where** <feature ...>, the <system name> shall <system response>.

For application configuration options, use the clauses like this:

`CONFIGURE_MAXIMUM_PROCESSORS` equal to one

> **Where** the system was configured with a processor maximum of exactly one, ...

`CONFIGURE_MAXIMUM_PROCESSORS` greater than one

> **Where** the system was configured with a processor maximum greater than one, ...

Please have a look at the following example used to specify `rtems_signal_catch()`. The example is a post-condition state specification of an action requirement, so there is an implicit set of pre-conditions and the trigger:

> **While** <pre-condition(s)>, **when** rtems_signal_catch() is called, ...

The *where* clauses should be mentally placed before the *while* clauses.

```
post-conditions:
- name: ASRInfo
  states:
  - name: NopNoPreempt
    test-code: |
      if ( rtems_configuration_get_maximum_processors() > 1 ) {
        CheckNoASRChange( ctx );
      } else {
        CheckNewASRSettings( ctx );
      }
    text: |
      Where the scheduler does not support the no-preempt mode, the ASR
      information of the caller of ${../if/catch:/name} shall not be
      changed by the ${../if/catch:/name} call.

      Where the scheduler does support the no-preempt mode, the ASR
      processing for the caller of ${../if/catch:/name} shall be done using
      the handler specified by ${../if/catch:/params[0]/name} in the mode
      specified by ${../if/catch:/params[1]/name}.
```

### 5.6.9 Action Requirements

*Action Requirement Item Type* (page 49) items may be used to specify and validate directive calls. They are a generator for event-driven requirements. Event-driven requirements should be written in the following *syntax* (page 19):

> **While** <pre-condition 0>, **while** <pre-condition 1>, ..., **while** <pre-condition *n*>, **when** <trigger>, the <system name> shall <system response>.

The list of *while* <pre-condition *i*> clauses for *i* from 1 to *n* in the EARS notation is generated by *n* pre-condition states in the action requirement item, see the `pre-condition` attribute in the *Action Requirement Item Type* (page 49).

The <trigger> in the EARS notation is defined for an action requirement item by the link to an SpecTypeInterfaceFunctionItemType or an SpecTypeInterfaceMacroItemType item using the *Interface Function Link Role* (page 85). The code provided by the `test-action` attribute defines the action code which should invoke the trigger directive in a particular set of pre-condition states.

Each post-condition state of the action requirement item generates a <system name> shall <system response> clause in the EARS notation, see the `post-condition` attribute in the *Action Requirement Item Type* (page 49).

Each entry in the transition map is an event-driven requirement composed of the pre-condition states, the trigger defined by the link to a directive, and the post-condition states. The transition map is defined by a list of *Action Requirement Transition* (page 65) descriptors.

Use `CamelCase` for the pre-condition names, post-condition names, and state names in action requirement items. The more conditions a directive has, the shorter should be the names. The transition map may be documented as a table and more conditions need more table columns. Use item attribute references in the `text` attributes. This allows context-sensitive substitutions.

#### 5.6.9.1 Example

Lets have a look at an example of an action requirement item. We would like to specify and validate the behaviour of the

```
rtems_status_code rtems_timer_create( rtems_name name, rtems_id *id );
```

directive which is particularly simple. For a more complex example see the specification of `rtems_signal_catch()` or `rtems_signal_send()` in `spec:/rtems/signal/req/catch` or `spec:/rtems/signal/send` respectively.

The event triggers are calls to `rtems_timer_create()`. Firstly, we need the list of pre-conditions relevant to this directive. Good candidates are the directive parameters, this gives us the `Name` and `Id` conditions. A system condition is if an inactive timer object is available so that we can create a timer, this gives us the `Free` condition. Secondly, we need the list of post-conditions relevant to this directive. They are the return status of the directive, `Status`, the validity of a unique object name, `Name`, and the value of an object identifier variable, `IdVar`. Each condition has a set of states, see the YAML data below for the details. The specified conditions and states yield the following transition map:

| Entry | Descriptor | Name | Id | Free | Status | Name | IdVar |
|---|---|---|---|---|---|---|---|
| 0 | 0 | Valid | Valid | Yes | Ok | Valid | Set |

continues on next page

Table 5.1 – continued from previous page

| Entry | Descriptor | Name | Id | Free | Status | Name | IdVar |
|---|---|---|---|---|---|---|---|
| 1 | 0 | Valid | Valid | No | TooMany | Invalid | Nop |
| 2 | 0 | Valid | Null | Yes | InvAddr | Invalid | Nop |
| 3 | 0 | Valid | Null | No | InvAddr | Invalid | Nop |
| 4 | 0 | Invalid | Valid | Yes | InvName | Invalid | Nop |
| 5 | 0 | Invalid | Valid | No | InvName | Invalid | Nop |
| 6 | 0 | Invalid | Null | Yes | InvName | Invalid | Nop |
| 7 | 0 | Invalid | Null | No | InvName | Invalid | Nop |

Not all transition maps are that small, the transition map of `rtems_task_mode()` has more than 8000 entries. We can construct requirements from the clauses of the entries. For example, the three requirements of entry 0 (Name=Valid, Id=Valid, and Free=Yes results in Status=Ok, Name=Valid, and IdVar=Set) are:

> While the `name` parameter is valid, while the `id` parameter references an object of type rtems_id, while the system has at least one inactive timer object available, when rtems_timer_create() is called, the return status of rtems_timer_create() shall be RTEMS_SUCCESSFUL.
>
> While the `name` parameter is valid, while the `id` parameter references an object of type rtems_id, while the system has at least one inactive timer object available, when rtems_timer_create() is called, the unique object name shall identify the timer created by the rtems_timer_create() call.
>
> While the `name` parameter is valid, while the `id` parameter references an object of type rtems_id, while the system has at least one inactive timer object available, when rtems_timer_create() is called, the value of the object referenced by the `id` parameter shall be set to the object identifier of the created timer after the return of the rtems_timer_create() call.

Now we will have a look at the specification item line by line. The top-level attributes are normally in alphabetical order in an item file. For this presentation we use a structured order.

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
copyrights:
- Copyright (C) 2021 embedded brains GmbH & Co. KG
enabled-by: true
functional-type: action
rationale: null
references: []
requirement-type: functional
```

The specification items need a bit of boilerplate to tell you what they are, who wrote them, and what their license is.

```
text: ${.:text-template}
```

Each requirement item needs a `text` attribute. For the action requirements, we do not have a single requirement. There is just a template indicator and no plain text. Several event-driven requirements are defined by the pre-conditions, the trigger, and the post-conditions.

```
pre-conditions:
- name: Name
  states:
  - name: Valid
    test-code: |
      ctx->name = NAME;
    text: |
      While the ${../if/create:/params[0]/name} parameter is valid.
  - name: Invalid
    test-code: |
      ctx->name = 0;
    text: |
      While the ${../if/create:/params[0]/name} parameter is invalid.
  test-epilogue: null
  test-prologue: null
- name: Id
  states:
  - name: Valid
    test-code: |
      ctx->id = &ctx->id_value;
    text: |
      While the ${../if/create:/params[1]/name} parameter references an object
      of type ${../../type/if/id:/name}.
  - name: 'Null'
    test-code: |
      ctx->id = NULL;
    text: |
      While the ${../if/create:/params[1]/name} parameter is
      ${/c/if/null:/name}.
  test-epilogue: null
  test-prologue: null
- name: Free
  states:
  - name: 'Yes'
    test-code: |
      /* Ensured by the test suite configuration */
    text: |
      While the system has at least one inactive timer object available.
  - name: 'No'
    test-code: |
      ctx->seized_objects = T_seize_objects( Create, NULL );
    text: |
      While the system has no inactive timer object available.
  test-epilogue: null
  test-prologue: null
```

This list defines the pre-conditions. Each pre-condition has a list of states and corresponding validation test code.

```
links:
```

(continues on next page)

(continued from previous page)

```
- role: interface-function
  uid: ../if/create
test-action: |
  ctx->status = rtems_timer_create( ctx->name, ctx->id );
```

The link to the `rtems_timer_create()` interface specification item with the `interface-function` link role defines the trigger. The `test-action` defines the how the triggering directive is invoked for the validation test depending on the pre-condition states. The code is not always as simple as in this example. The validation test is defined in this item along with the specification.

```
post-conditions:
- name: Status
  states:
  - name: Ok
    test-code: |
      T_rsc_success( ctx->status );
    text: |
      The return status of ${../if/create:/name} shall be
      ${../../status/if/successful:/name}.
  - name: InvName
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_NAME );
    text: |
      The return status of ${../if/create:/name} shall be
      ${../../status/if/invalid-name:/name}.
  - name: InvAddr
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_ADDRESS );
    text: |
      The return status of ${../if/create:/name} shall be
      ${../../status/if/invalid-address:/name}.
  - name: TooMany
    test-code: |
      T_rsc( ctx->status, RTEMS_TOO_MANY );
    text: |
      The return status of ${../if/create:/name} shall be
      ${../../status/if/too-many:/name}.
  test-epilogue: null
  test-prologue: null
- name: Name
  states:
  - name: Valid
    test-code: |
      id = 0;
      sc = rtems_timer_ident( NAME, &id );
      T_rsc_success( sc );
      T_eq_u32( id, ctx->id_value );
    text: |
      The unique object name shall identify the timer created by the
```

(continues on next page)

(continued from previous page)

```
      ${../if/create:/name} call.
  - name: Invalid
    test-code: |
      sc = rtems_timer_ident( NAME, &id );
      T_rsc( sc, RTEMS_INVALID_NAME );
    text: |
      The unique object name shall not identify a timer.
  test-epilogue: null
  test-prologue: |
    rtems_status_code sc;
    rtems_id          id;
- name: IdVar
  states:
  - name: Set
    test-code: |
      T_eq_ptr( ctx->id, &ctx->id_value );
      T_ne_u32( ctx->id_value, INVALID_ID );
    text: |
      The value of the object referenced by the ${../if/create:/params[1]/name}
      parameter shall be set to the object identifier of the created timer
      after the return of the ${../if/create:/name} call.
  - name: Nop
    test-code: |
      T_eq_u32( ctx->id_value, INVALID_ID );
    text: |
      Objects referenced by the ${../if/create:/params[1]/name} parameter in
      past calls to ${../if/create:/name} shall not be accessed by the
      ${../if/create:/name} call.
  test-epilogue: null
  test-prologue: null
```

This list defines the post-conditions. Each post-condition has a list of states and corresponding validation test code.

```
skip-reasons: {}
transition-map:
- enabled-by: true
  post-conditions:
    Status:
    - if:
        pre-conditions:
          Name: Invalid
      then: InvName
    - if:
        pre-conditions:
          Id: 'Null'
      then: InvAddr
    - if:
        pre-conditions:
```

(continues on next page)

(continued from previous page)

```
          Free: 'No'
      then: TooMany
    - else: Ok
    Name:
    - if:
        post-conditions:
          Status: Ok
      then: Valid
    - else: Invalid
    IdVar:
    - if:
        post-conditions:
          Status: Ok
      then: Set
    - else: Nop
  pre-conditions:
    Name: all
    Id: all
    Free: all
type: requirement
```

This list of transition descriptors defines the transition map. For the post-conditions, you can use expressions to ease the specification, see *Action Requirement Transition Post-Condition State* (page 66). The `skip-reasons` can be used to skip entire entries in the transition map, see *Action Requirement Skip Reasons* (page 65).

```
test-brief: null
test-description: null
```

The item contains the validation test code. The validation test in general can be described by these two attributes.

```
test-target: testsuites/validation/tc-timer-create.c
```

This is the target file for the generated validation test code. Make sure this file is included in the build specification, otherwise the test code generation will fail.

```
test-includes:
- rtems.h
- string.h
test-local-includes: []
```

You can specify a list of includes for the validation test.

```
test-header: null
```

A test header may be used to create a parameterized validation test, see *Test Header* (page 108). This is an advanced topic, see the specification of `rtems_task_ident()` for an example.

```
test-context-support: null
test-context:
```

(continues on next page)

(continued from previous page)

```
- brief: |
    This member is used by the T_seize_objects() and T_surrender_objects()
    support functions.
  description: null
  member: |
    void *seized_objects
- brief: |
    This member may contain the object identifier returned by
    rtems_timer_create().
  description: null
  member: |
    rtems_id id_value
- brief: |
    This member specifies the ${../if/create:/params[0]/name} parameter for the
    action.
  description: null
  member: |
    rtems_name name
- brief: |
    This member specifies the ${../if/create:/params[1]/name} parameter for the
    action.
  description: null
  member: |
    rtems_id *id
- brief: |
    This member contains the return status of the action.
  description: null
  member: |
    rtems_status_code status
```

You can specify a list of validation test context members which can be used to maintain the state of the validation test. The context is available through an implicit `ctx` variable in all code blocks except the support blocks. The context support code can be used to define test-specific types used by context members. Do not use global variables.

```
test-support: |
  #define NAME rtems_build_name( 'T', 'E', 'S', 'T' )

  #define INVALID_ID 0xffffffff

  static rtems_status_code Create( void *arg, uint32_t *id )
  {
    return rtems_timer_create( rtems_build_name( 'S', 'I', 'Z', 'E' ), id );
  }
```

The support code block can be used to provide functions, data structures, and constants for the validation test.

```
test-prepare: null
test-cleanup: |
```

(continues on next page)

(continued from previous page)

```
  if ( ctx->id_value != INVALID_ID ) {
    rtems_status_code sc;

    sc = rtems_timer_delete( ctx->id_value );
    T_rsc_success( sc );

    ctx->id_value = INVALID_ID;
  }

  T_surrender_objects( &ctx->seized_objects, rtems_timer_delete );
```

The validation test basically executes a couple of nested for loops to iterate over each pre-condition and each state of the pre-conditions. These two optional code blocks can be used to prepare the pre-condition state preparations and clean up after the post-condition checks in each loop iteration.

```
test-setup:
  brief: null
  code: |
    memset( ctx, 0, sizeof( *ctx ) );
    ctx->id_value = INVALID_ID;
  description: null
test-stop: null
test-teardown: null
```

These optional code blocks correspond to test fixture methods, see *Test Fixture* (page 183).

#### 5.6.9.2 Pre-Condition Templates

Specify all directive parameters as separate pre-conditions. Use the following syntax for directive object identifier parameters:

```
- name: Id
  states:
  - name: NoObj
    test-code: |
      ctx->id = 0xffffffff;
    text: |
      While the ${../if/directive:/params[0]/name} parameter is not
      associated with a thing.
  - name: ClassA
    test-code: |
      ctx->id = ctx->class_a_id;
    text: |
      While the ${../if/directive:/params[0]/name} parameter is associated
      with a class A thing.
  - name: ClassB
    test-code: |
      ctx->id = ctx->class_b_id;
    text: |
```

(continues on next page)

(continued from previous page)

```
      While the ${../if/directive:/params[0]/name} parameter is associated
      with a class B thing.
  test-epilogue: null
  test-prologue: null
```

Do not add specifications for invalid pointers. In general, there are a lot of invalid pointers and the use of an invalid pointer is in almost all cases undefined behaviour in RTEMS. There may be specifications for special cases which deal with some very specific invalid pointers such as the `NULL` pointer or pointers which do not satisfy a range or boundary condition. Use the following syntax for directive pointer parameters:

```
- name: Id
  states:
  - name: Valid
    test-code: |
      ctx->id = &ctx->id_value;
    text: |
      While the ${../if/directive:/params[3]/name} parameter references an
      object of type ${../../type/if/id:/name}.
  - name: 'Null'
    test-code: |
      ctx->id = NULL;
    text: |
      While the ${../if/directive:/params[3]/name} parameter is
      ${/c/if/null:/name}.
  test-epilogue: null
  test-prologue: null
```

Use the following syntax for other directive parameters:

```
- name: Name
  states:
  - name: Valid
    test-code: |
      ctx->name = NAME;
    text: |
      While the ${../if/directive:/params[0]/name} parameter is valid.
  - name: Invalid
    test-code: |
      ctx->name = 0;
    text: |
      While the ${../if/directive:/params[0]/name} parameter is invalid.
  test-epilogue: null
  test-prologue: null
```

#### 5.6.9.3 Post-Condition Templates

Do not mix different things into one post-condition. If you write multiple sentences to describe what happened, then think about splitting up the post-condition. Keep the post-condition simple and focus on one testable aspect which may be changed by a directive call.

For directives returning an `rtems_status_code` use the following post-condition states. Specify only status codes which may be returned by the directive. Use it as the first post-condition. The first state shall be `Ok`. The other states shall be listed in the order in which they can occur.

```
- name: Status
  states:
  - name: Ok
    test-code: |
      T_rsc_success( ctx->status );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/successful:/name}.
  - name: IncStat
    test-code: |
      T_rsc( ctx->status, RTEMS_INCORRECT_STATE );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/incorrect-state:/name}.
  - name: InvAddr
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_ADDRESS );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/invalid-address:/name}.
  - name: InvName
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_NAME );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/invalid-name:/name}.
  - name: InvNum
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_NUMBER );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/invalid-number:/name}.
  - name: InvSize
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_SIZE );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/invalid-size:/name}.
  - name: InvPrio
    test-code: |
      T_rsc( ctx->status, RTEMS_INVALID_PRIORITY );
    text: |
```

(continues on next page)

(continued from previous page)

```
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/invalid-priority:/name}.
  - name: NotConf
    test-code: |
      T_rsc( ctx->status, RTEMS_NOT_CONFIGURED );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/not-configured:/name}.
  - name: NotDef
    test-code: |
      T_rsc( ctx->status, RTEMS_NOT_DEFINED );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/not-defined:/name}.
  - name: NotImpl
    test-code: |
      T_rsc( ctx->status, RTEMS_NOT_IMPLEMENTED );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/not-implemented:/name}.
  - name: TooMany
    test-code: |
      T_rsc( ctx->status, RTEMS_TOO_MANY );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/too-many:/name}.
  - name: Unsat
    test-code: |
      T_rsc( ctx->status, RTEMS_UNSATISFIED  );
    text: |
      The return status of ${../if/directive:/name} shall be
      ${../../status/if/unsatisfied:/name}.
  test-epilogue: null
  test-prologue: null
```

For values which are returned by reference through directive parameters, use the following post-condition states.

```
- name: SomeParamVar
  states:
  - name: Set
    test-code: |
      /* Add code to check that the object value was set to X */
    text: |
      The value of the object referenced by the
      ${../if/directive:/params[0]/name} parameter shall be set to X after
      the return of the ${../if/directive:/name} call.
  - name: Nop
    test-code: |
```

(continues on next page)

(continued from previous page)

```
      /* Add code to check that the object was not modified */
    text: |
      Objects referenced by the ${../if/directive:/params[0]/name}
      parameter in past calls to ${../if/directive:/name} shall not be
      accessed by the ${../if/directive:/name} call.
```

### 5.6.10 Validation Test Guidelines

The validation test cases, test runners, and test suites are generated by the `./spec2modules.py` script from specification items. For the placement and naming of the generated sources use the following rules:

- Place architecture-specific validation test sources and programs into the `testsuites/validation/cpu` directory.
- Place BSP-specific validation test sources and programs into the `testsuites/validation/bsps` directory.
- Place all other validation test sources and programs into the `testsuites/validation` directory.
- Place architecture-specific unit test sources and programs into the `testsuites/unit/cpu` directory.
- Place BSP-specific unit test sources and programs into the `testsuites/unit/bsps` directory.
- Place all other unit test sources and programs into the `testsuites/unit` directory.
- Use dashes (`-`) to separate parts of a file name. Use only dashes, the digits `0` to `9`, and the lower case characters `a` to `z` for file names. In particular, do not use underscores (`_`).
- The parts of a file name shall be separated by dashes and ordered from most general (left) to more specific (right), for example `tc-task-construct.c`.
- The file names associated with tests shall be unique within the system since the test framework prints out only the base file names.
- Use the prefix `tc-` for test case files.
- Use the prefix `tr-` for test runner files.
- Use the prefix `ts-` for test suite files.
- Use the prefix `tx-` for test extension files (test support code).
- Tests for fatal errors shall have `fatal` as the most general file part, for example `ts-fatal-too-large-tls-size.c`.
- Validation test suites shall have `validation` as the most general file part, for example `ts-validation-no-clock-0.c`.
- Unit test suites shall have `unit` as the most general file part, for example `ts-unit-no-clock-0.c`.
- Architecture-specific files shall have the architecture name as a file part, for example `ts-fatal-sparc-leon3-clock-initialization.c`.

- BSP-specific files shall have the BSP family or variant name as a file part, for example `tc-sparc-gr712rc.c`.
- Architecture-specific or BSP-specific tests shall use the `enabled-by` attribute of the associated specification item to make the build item conditional, for example:

```
...
build-type: objects
enabled-by: arm
type: build
...
```

```
...
build-type: test-program
enabled-by: bsps/sparc/leon3
type: build
...
```

### 5.6.11 Verify the Specification Items

The `./specverify.py` script verifies that the specification items have the format documented in *Specification Items* (page 24). To some extent the values of attributes are verified as well.

CHAPTER

**SIX**

# SOFTWARE DEVELOPMENT MANAGEMENT

## 6.1 Software Development with GitLab

### 6.1.1 Browse the Git Repository Online

You can browse repositories from your home page or from https://gitlab.rtems.org/explore/projects to see a global view including all forks and most starred projects on the RTEMS gitlab instance.

The sort order can be changed by the drop-down at the far right of the GUI. If you are logged in, your preferences should be saved so that the sort order is persistent.

From your home page, you can also view your starred repositories. This is a handy page to use to navigate to the repositories that interest you the most. In addition, you can change your preferences for your Homepage to show the Starred Projects among other possibly useful landing pages.

### 6.1.2 Using Git

An exhaustive treatment here is not possible. We suggest availing yourself of online resources to learn how to use Git, such as:

- http://gitready.com/ - An excellent resource from beginner to very advanced.
- https://git-scm.com/docs - The official Git reference.
- https://docs.gitlab.com/ - GitLab's documentation.

### 6.1.3 Making Changes with Branches

Git allows you to make changes in the RTEMS source tree and track those changes locally. We recommend you make all your changes in local branches. If you are working on a few different changes or a progression of changes it is best to use a local branch for each change.

A branch for each change lets your repo's main branch track the upstream RTEMS' main branch without interacting with any of the changes you are working on. A completed change is submitted as Merge Request for review and this can take time. While this is happening the upstream's main branch may be updated and you may need to rebase your work and test again if you are required to change or update your patch. A local branch isolates a specific change from others and helps you manage the process.

### 6.1.4 Working with Remote Branches

The previous releases of RTEMS are available through remote branches. To check out a remote branch, first query the Git repository for the list of branches:

```
git branch origin -r
```

Then check out the desired remote branch, for example:

```
git checkout -t rtems70 origin/7.0
```

Or if you have previously checked out the remote branch then you should see it in your local branches:

```
git branch
```

### 6.1.5 Rebasing

An alternative to the merge command is rebase, which replays the changes (commits) on one branch onto another. `git rebase` finds the common ancestor of the two branches, stores each commit of the branch you are on to temporary files and applies each commit in order.

Rebasing makes a cleaner history than merging; the log of a rebased branch looks like a linear history as if the work was done serially rather than in parallel. A primary reason to rebase is to ensure commits apply cleanly on a remote branch, e.g. when submitting patches to RTEMS that you create by working on a branch in a personal repository. Using rebase to merge your work with the remote branch eliminates most integration work for the committer/maintainer. We require maintaining a linear history in our repositories.

There is one caveat to using rebase: Do not rebase commits that you have pushed to a public repository. Rebase abandons existing commits and creates new ones that are similar but different. If you push commits that others pull down, and then you rewrite those commits with `git rebase` and push them up again, the others will have to re-merge their work and trying to integrate their work into yours can become messy.

### 6.1.6 Commit Message Guidance

The commit message associated with a change to any software project is of critical importance. It is the explanation of the change and the rationale for it. Future users looking back through the project history will rely on it. Even the author of the change will likely rely on it once they have forgotten the details of the change. It is important to make the message useful. Here are some guidelines followed by the RTEMS Project to help improve the quality of our commit messages.

- When committing a change the first line is a summary. Please make it short while hinting at the nature of the change. You can discuss the change if you wish in a ticket that has a PR number which can be referenced in the commit message. After the first line, leave an empty line and add whatever required details you feel are needed.
- Patches should be as single purpose as possible. This is reflected in the first line summary message. If you find yourself writing something like “Fixed X and Y”, “Updated A and B”, or similar, then evaluate whether the patch should really be a patch series rather than a single larger patch.
- Format the commit message so it is readable and clear. If you have specific points related to the change make them with separate paragraphs and if you wish you can optionally uses a - marker with suitable indents and alignment to aid readability.
- Limit the line length to less than 80 characters
- Please use a real name with a valid email address. Please do not use pseudonyms or provide anonymous contributions.
- Please do not use terms such as “Fix bug”, “With this change it works”, or “Bump hash”. If you fix a bug please state the nature of the bug and why this change fixes it. If a change makes something work then detail the reason. You do not need to explain the change line by line as the commits diff and associated ticket will.
- If you change the formatting of source code in a repository please make that a separate patch and use “Formatting changes only” on the first line. Please indicate the reason or process. For example to “Conforming to code standing”, “Reverting to upstream format”, “Result of automatic formatting”.

- Similarly, if addressing a spelling, grammar, or Doxygen issue, please put that in a commit by itself separate from technical changes.

An example commit message:

```
test/change: Test message on formatting of commits

- Shows a simple single first line

- Has an empty second line

- Shows the specifics of adding separate points in the commit message as
  separate paragraphs. It also shows a `-` separator and multilines
  that are less than the 80 character width

- Show a ticket update and close

Updates #9876
Closes #8765
```

The first line generally starts with a file or directory name which indicates the area in RTEMS to which the commit applies. For a patch series which impacts multiple BSPs, it is common to put each BSP into a separate patch. This improves the quality and specificity of the commit messages.

### 6.1.7 Preparing and Submitting Merge Requests

The RTEMS Project uses Git for version control and uses Gitlab for managing changes. Contributions are made by creating a Merge Request (MR) on Gitlab. The Gitlab merge request documentation comprehensively explains the concepts of using merge requests. RTEMS will only accept changes via a Merge Request. Most merge requests should have one or more Issues associated with them, unless it is a simple, isolated change. Please do not use merge requests to introduce new code, concepts, styles or to change existing behaviours such as APIs or internal interfaces. Please create an issue before you start the work so the community is aware of the changes coming. The merge requests can then focus on the details of the implementation and approval does not need to be about approval of change at a functional level.

We use project forks as the base of our workflow and outside of that there is no workflow we mandate. What works for one task or work package may not work for another. Complex tasks may affect a number of our GitLab Projects with issues and merge requests in a number of projects. You may want to use an Epic to bring work together.

With our GitLab instance, you fork a repo into your personal workspace and use that to manage your changes. This means you need to keep your forked project up to date. See the Gitlab forking workflow documentation for details. If you are part of a team working on a change you can collaborate on merge requests. GitLab enforces branch naming rules and provides branch naming patterns that simplifies code review and software change management. You can create merge requests from your fork back to the upstream repository. We do not normally squash merge requests. A merge request with more than one commit should be buildable at each commit so a bisect of main does not break.

#### 6.1.7.1 Checklist for Merge Requests

Check the following items before you publish your merge requests:

- The author name of each commit is a full name of the author.
- The author email of each commit is a valid email address for the author.
- The licence conditions of the contributed content allow an integration into the RTEMS code base.
- If you are the copyright holder of the entire patch content, then please contribute it under the BSD-2-Clause license. For documentation use CC BY-SA 4.0.
- Make sure you have a meaningful title which does not exceed 50 characters in one line. Use a "topic: The meaningful title" style. A topic could be the main component of the commit. Just have a look at existing commit messages.
- Each patch has a good commit message. It should describe the reason for the change. It should list alternative approaches and why they were not chosen.
- The code changes honour the coding style. At least do your changes in the style of the surrounding code.
- Each patch contains no spelling mistakes and grammar errors.
- Each patch is easy to review. It changes one thing only and contains no unrelated changes. Format changes should be separated from functional changes.
- If commits correspond to Issues, the merge request should have "Close #X." or "Update #X." to update status once it is merged.
- Each patch builds. All RTEMS tests link with every patch.
- Each patch does not introduce new compiler warnings.
- Each patch does not introduce new test failures in existing tests.

#### 6.1.7.2 Updating a Merge Request

As you make changes to your merge request through the review process, you will either be layering additional patches on top of your current patch set, or you will be rebasing your patch set. Either approach is acceptable, but remember that every patch in your patch set must pass continuous integration testing and adhere to the *Checklist for Merge Requests* (page 143). Most often, this means that you should rebase the patch set in your merge request to include the updates.

There are several ways you can rebase the patch set. The following is one suggested workflow:

- Before making changes, create a new local branch of your merge request. Make your changes on this branch, so that you can always go back to your previous state. Always keep your original branch until you have pushed a new, clean version that supersedes it. Even then, you may want to keep your original branch around in case something went wrong that you did not notice, such as you accidentally removed a necessary commit while rebasing.
- Make and commit changes locally until you are satisfied with your code
- Interactively rebase your local branch using `git rebase --interactive` to allow you to select the order of commits and to reword or fixup commits. One good strategy here is to reorder and fixup commits in one round and then reword them in a second round, so that

you get your commits in the right order and shape you want before finalizing the patch descriptions.

- Force-push your local branch to your merge request branch on your fork. If something goes wrong, you can revert back to your local version.

### 6.1.8 Rebasing a Merge Request

You can follow a similar process as *Updating a Merge Request* (page 143) to rebase your merge request branch to an updated target branch, e.g., to pick up changes on `main`. In this case, after creating a local branch, use `git pull --rebase` stopping to fix merge conflicts along the way. If it gets out of hand, you can either `abort` the rebase or you can go back to your original branch.

When merge conflicts are too much to handle doing a rebase, you may instead like to create a fresh branch from `main` and then use `git-cherry-pick` to pull commits from your merge request branch on to the head of `main`. If you are having too much trouble, ask for help.

### 6.1.9 Merge Request Review Process

Merge requests sent to the RTEMS Gitlab undergo a public review process. At least two approvals are required before a merge request can be pushed to the RTEMS repository. It helps if you follow the *Checklist for Merge Requests* (page 143). An easy to review patch series which meets the quality standards of the RTEMS Project will be more likely to get integrated quickly.

The review process includes both objective and subjective feedback. You should reflect upon and consider all feedback before making or refusing changes. It is important to note that some feedback may be relevant at one point in time, but less relevant in the future. Also, what concerns one developer may not concern another. Just because you address all the feedback in one round of review does not mean your submission will be approved, as someone (even the same reviewer) may notice something that was not seen before. It is important to have patience, humility, and open-mindedness when engaging in open-source software review and approval processes. This is true for both contributors and reviewers.

Reviews should be conducted primarily via the GitLab interface using the in-line commenting feature. Follow-up comments to the same line should be threaded, while new comments should be added to the specific, relevant line of modified code. Although high-level comments about the entire patch set are allowed, the most useful commments are those that are specifically targeted to problematic lines of code.

Threads usually should be resolved by the first author of the thread, i.e., the reviewer who wrote the first comment. In particular, code submitters should not resolve the threads but should reply to the thread to indicate that they think they have addressed the concern.

#### 6.1.9.1 Approvers

Merge Request approvals must be from a code owner, identified by the `CODEOWNERS` file and by sub-groups beneath `Approvers`. Any one who has requested approval permission can approve a merge request. Once a patch series is approved for integration into the RTEMS code base it can be merged by anyone with merge rights, which may include an automated bot.

Approvers are volunteering their time so be polite. If you do not get a response to a merge request after five working days, please send a reminder on the merge request or send email to the Developers Mailing List.

## 6.2 Change Control with GitLab

### 6.2.1 Updating GitLab Metadata

Only GitLab accounts with a role of Developer or above have permissions to modify the metadata of Issues and Merge Requests. This section is intended for those individuals.

#### 6.2.1.1 Assignee

The assignee of a merge request should usually be set to the submitter. Not all users will appear in the Edit boxes. Add the user by a @user in a comment, refresh your browser and then you can edit the assignee.

The assignee of an Issue should be set to someone (accepting to be) responsible for handling it.

#### 6.2.1.2 Reviewers

You may add reviewers to a Merge Request to ensure specific individuals are in the approval chain. This should be done (or undone) at your discretion.

#### 6.2.1.3 Labels

Until further guidance can be determined, use your best effort to assign one or more relevant labels to Issues and Merge Requests.

#### 6.2.1.4 Milestones

The milestone assigned to a Merge Request must match the branch that it targets. Usually, the target branch should be `main` and you should set the milestone to the next release. See below in case a Merge Request targets a release branch.

#### 6.2.1.5 Release Branches

There are additional checks to use for any changes that directly apply or are backported to a release branch. These checks need to satisfy the requirements of the *Release Process* (page 264). For requested backports, please refer directly to *How to Handle Backports* (page 274). With any Merge Request targeting a release branch, ensure that:

- The commit messages of the Merge Request refer to an Issue with the Milestone that matches the release branch. Add this Milestone to the Merge Request.
- The Issue referenced by the Merge Request is linked to the Epic for the target release.

### 6.2.2 Approving Merge Requests

This section is intended for those individuals who are `Approvers` and for anyone else who wants to participate in the *Merge Request Review Process* (page 144) as a code reviewer.

#### 6.2.2.1 Conflict of Interest Rules

We do not have formally adopted Conflict of Interest rules, but we generally avoid public review and approval by individuals who (should) have privately reviewed code. This means that you should not approve code that is submitted by someone who works with you commercially. Instead, you could provide input to the review or to assist your colleague in resolving review feedback.

#### 6.2.2.2 Continuous Integration Pipelines

Check the status of the pipelines and give feedback to help contributors to fix errors.

#### 6.2.2.3 Check the Commits

Ensure the commits are properly made according to the *Commit Message Guidance* (page 141). For new contributors, download the patch file (under the Code button) and make sure they have set a name and email address in their git metadata.

#### 6.2.2.4 Creating a Review

Navigate to the `Changes` tab in GitLab, and provide inline comments. It is up to you if you want to add comments one at a time or in bulk using the "Start a review" option.

Avoid adding comments directly in the `Overview` tab. If you do make comments there that need to be addressed, you should ensure they show up as "threads" by either checking that option when you write the comment, or by replying to your own comment to turn it into a thread.

Each comment that you add should only be on one concern. Grouping multiple review concerns in the same comment makes it harder to check that all concerns are resolved satisfactorily.

#### 6.2.2.5 Marking a Merge Request as a Draft

You can mark a Merge Request as a Draft if there are deficiencies that will take some time to resolve or simply to prevent it from being merged. In general, other reviewers will ignore such Draft Merge Requests. This should be used to reduce reviewer burden or to indicate the change should be held off regardless of the approval. For example, during the preparation of a release the pending Merge Requests may be marked as Draft to prevent them from being merged.

#### 6.2.2.6 Cloning a Merge Request

Prior to approving, it is your responsibility to ensure that the change works as intended. In some cases this may be done without having to clone and test changes, but often you will need to do some work locally. There are a few ways you can do this. The GitLab Documentation has advice on this. You can also get a check-out from the "Code" button dropdown, "Check out branch" menu.

The following are some potentially useful `git` aliases that may help simplify this process:

```
[alias]
  mr = !sh -c 'git fetch $1 merge-requests/$2/head:mr-$1-$2 && git checkout mr-$1-$2' -
  fetch-fork = !sh -c 'git remote add $1 ssh://git@gitlab.rtems.org:2222/$1/$(basename $(git rev-parse --show-toplevel)).git && git fetch $1' -
  push-fork = !sh -c 'git push $1 $(git rev-parse --abbrev-ref --symbolic-full-name HEAD):$2 $3' -
  checkout-fork = !git checkout -t $1/$2
  close-fork = !git remote remove
  close-mr = !sh -c 'git checkout main && git fetch && git pull && git branch -d mr-$1-$2' -
```

These aliases were created on a Linux machine and would likely need modification in other hosts. The mainly useful alias is the first one, you can use `git mr origin 123` to checkout

merge request 123 against the origin repo. If you want to make changes and push them back, you have to use the other commands, for example:

```
git fetch-fork gedare
git checkout-fork gedare somebranch
edit edit edit
git commit/rebase
git push-fork gedare somebranch -f
```

Which is a fairly rude thing to do, but can be a handy way to fix a submitted Merge Request if the submitter is unable to do so themselves. The ability to push to the Merge Request submitter's branch can be disabled, but it is turned on by default.

#### 6.2.2.7 Approving

Approve! Merge! If you think it should be merged but cannot, check for reasons that block it. For example, if it just needs to be rebased, do it! If the rebase has conflicts, you can either ask the submitter to handle it, or you can use the preceding instructions for cloning the Merge Request to fix it yourself. It is generally preferable to give the submitter a chance to fix it on their own. If you cannot see why a Merge Request cannot be merged, please make an inquiry in the Discord.

## 6.3 Coding Standards

TBD - Write introduction, re-order, identify missing content

### 6.3.1 Coding Conventions

The style of RTEMS is generally consistent in the core areas. This section attempts to capture generally accepted practices. When in doubt, consult the code around you, look in the RTEMS sources in the directories `cpukit/include/rtems/score` and `cpukit/score`, or ask on the Developers Mailing List.

#### 6.3.1.1 Coding Style

See *Code Formatting* (page 153).

#### 6.3.1.2 Source Documentation

- Use Doxygen according to our *Doxygen Guidelines* (page 157).
- Use the file templates, see *File Templates* (page 163).
- Use `/* */` comments.
- Do not use `//` comments.
- Use comments wisely within function bodies, to explain or draw attention without being verbose.
- Use English prose and strive for good grammar, spelling, and punctuation.
- Use `TODO` with a comment to indicate code that needs improvement. Make it clear what there is to do. Add a ticket and add a link to it.
- Use `XXX` or `FIXME` to indicate an error/bug/broken code. Add a ticket and add a link to it.

#### 6.3.1.3 Licenses

The RTEMS Project has strict requirements on the types of software licenses that apply to software it includes and distributes. Submissions will be summarily rejected that do not follow the correct license or file header requirements.

- Refer to *Licensing Requirements* (page 279) for a discussion of the acceptable licenses and the rationale.
- Refer to *Copyright and License Block* (page 163) for example copyright/license comment blocks for various languages.

#### 6.3.1.4 Third-Party Source Code

The appropriate use of code from other open-source projects is encouraged. We refer to such code as “third-party code” and we refer to the origin project as the “upstream” source. We treat third-party code carefully to ensure compliance with license terms and to ease maintenance burdens. We aim to return code back to the upstream whenever possible. The following guidelines should be followed to meet the high-level goal of respecting the third-party code and upstream.

When importing code from anywhere you must retain the original code’s licensing and copyright or other attribution information. Be careful with copyright and code ownership, these things matter. The best approach is to provide an isolated patch that adds all of the code from the

third party, and then layer on patches that modify or make use of the third party code. Attempt to minimize changes, and submit patches upstream when possible.

When you have to change third-party code, it is best to provide a clear identification of the change like this, omitting the comments:

```
/* unmodified code */
#if defined(__rtems__)
  /* changes made */
#endif
/* unmodified code */
```

This approach helps to minimize code review, identify very clearly the origin of source code, and eases maintenance in case of updating the third-party code.

- Exception: unmaintained third-party code adopted and maintained by RTEMS may be directly modified and reformatted to a suitable style and to meet coding conventions.

#### 6.3.1.5 Language and Compiler

- Use C99.
- Treat warnings as errors: eliminate them.
- Favor C, but when assembly language is required use inline assembly if possible.
- Do not use compiler extensions.
- Use the RTEMS macros defined in <rtems/score/basedefs.h> for abstracting compiler-specific features. For using attributes see the GCC attribute syntax. Prefer to place attributes in front of the declarator. Try to be in line with C++11 attributes and C11 keywords such as _Noreturn.
- Use NULL for the null pointer, and prefer to use explicit checks against NULL, e.g.,

```
if ( ptr != NULL )
```

  instead of

```
if ( !ptr )
```

- Use explicit checks for bits in variables.
  - Example 1: Use

```
if ( XBITS == (var & XBITS) )
```

    to check for a set of defined bits.
  - Example 2: Use

```
if ( (var & X_FLAGS) != 0) )
```

    instead of

```
if ( !!(var & X_FLAGS) )
```

    to check for at least 1 defined bit in a set.

- Use `(void) unused;` to mark unused parameters and set-but-unused variables immediately after being set.
- Do not put function prototypes in C source files, any global functions should have a prototype in a header file and any private function should be declared static.
- Declare global variables in exactly one header file. Define global variables in at most one source file. Include the header file declaring the global variable as the first include file if possible to make sure that the compiler checks the declaration and definition and that the header file is self-contained.
- Do not cast arguments to any printf() or printk() variant. Use <inttypes.h> PRI constants for the types supported there. Use <rtems/inttypes.h> for the other POSIX and RTEMS types that have PRI constants defined there. This increases the portability of the printf() format.
- Do not use the register keyword. It is deprecated since C++14.

#### Compile-Time Conditional Code Features

Some RTEMS features are compile-time dependent and normally can be enabled/disabled via RTEMS build configuration options, for example `ENABLE_SMP`, `ENABLE_PROFILING`, etc. There usually exists a C pre-processor symbol which is defined in case the feature is enabled, e.g., `RTEMS_SMP`, `RTEMS_PROFILING`, etc. The following rules should be followed when using these conditional features:

- Use inline functions to wrap code-blocks controlled by conditional features.
- The inline function should evaluate to an empty body if the feature is not defined whenever possible.
- Use `(void) arg;` to silence unused parameter warnings within the function.

This provides type checks for the function calls even in case the feature is disabled. The compiler can easily optimize empty inline functions away. Example:

```
static inline feature_x_func(int a, double b, void *c)
{
  #ifdef FEATURE_X
    /* Do something */
  #else
    (void) a;
    (void) b;
    (void) c;
  #endif
}
```

#### 6.3.1.6 Readability

- Understand and follow the *Naming Rules* (page 168).
- Use typedef to remove ‘struct’, but do not use typedef to hide pointers or arrays. * Exception: typedef can be used to simplify function pointer types.
- Do not mix variable declarations and code.
- Declare variables at the start of a block.

- Only use primitive initialization of variables at their declarations. Avoid complex initializations or function calls in variable declarations.
- Do not put unrelated functions or data in a single file.
- Do not declare functions inside functions.
- Avoid deep nesting by using early exits e.g. return, break, continue. * Parameter checking should be done first with early error returns. * Avoid allocation and critical sections until error checking is done. * For error checks that require locking, do the checks early after acquiring locks. * Use of 'goto' requires good reason and justification.
- Test and action should stay close together.
- Avoid complex logic in conditional and loop statements.
- Put conditional and loop statements on the line after the expression.
- Favor inline functions to hide [compile-time conditional code features].
- Define non-inline functions in a .c source file.
- Declare all global (non-static) functions in a .h header file.
- Declare and define inline functions in one place. Usually, this is a *impl.h* header file.
- Declare and define static functions in one place. Usually, this is toward the start of a .c file. Minimize forward declarations of static functions.
- Function declarations should include variable names.
- Avoid excess parentheses. Learn the operator precedence rules.
- Always use parentheses with sizeof. This is an exception to the rule about excess parentheses.

#### 6.3.1.7 Robustness

- Check all return statuses.
- Validate input parameters.
- Use debug assertions (assert).
- Use const when appropriate for read-only function parameters and compile-time constant values.
- Do not hard code limits such as maximum instances into your code.
- Prefer to use sizeof(variable) instead of sizeof(type).
- Favor C automatic variables over global or static variables.
- Use global variables only when necessary and ensure atomicity of operations.
- Do not shadow variables.
- Avoid declaring large buffers or structures on the stack.
- Avoid using zero (0) as a valid value. Memory often defaults to being zero.
- Favor mutual exclusion primitives over disabling preemption.
- Avoid unnecessary dependencies, such as by not calling "printf()" on error paths.

- Avoid inline functions and macros with complicated logic and decision points.
- Prefer inline functions, enum, and const variables instead of CPP macros.
- CPP macros should use a leading underscore for parameter names and avoid macro pitfalls.

#### 6.3.1.8 Portability

- Think portable! RTEMS supports a lot of target hardware.
- For integer primitives, prefer to use precise-width integer types from C99 stdint.h.
- Write code that is 16-bit, 32-bit, and 64-bit friendly.

#### 6.3.1.9 Maintainability

- Minimize modifications to [third-party source code].
- Keep it simple! Simple code is easier to debug and easier to read than clever code.
- Share code with other architectures, CPUs, and BSPs where possible.
- Do not duplicate standard OS or C Library routines.

#### 6.3.1.10 Performance

- Prefer algorithms with the lowest order of time and space. for fast, deterministic execution times with small memory footprints.
- Understand the constraints of real-time programming.
  - Limit execution times in interrupt contexts and critical sections, such as Interrupt and Timer Service Routines (TSRs).
- Prefer to `++preincrement` instead of `postincrement++`.
- Avoid using floating point except where absolutely necessary.

#### 6.3.1.11 Miscellaneous

- If you need to temporarily change the execution mode of a task/thread, restore it.
- If adding code to "cpukit" be sure the filename is unique since all files under that directory get merged into a single library.

#### 6.3.1.12 Header Files

- Do not add top-level header files. Place the header files in a directory, for example `#include <rtems/*>`, `#include <bsp/*>`, `#include <dev/*>`, etc.
- Use the extension `.h` for C header files.
- Use the extension `.hpp` for C++ header files.
- Use the file template for header files, see *C/C++ Header File Template* (page 164).
- Use separate header files for the API and the implementation.
- Use `foobar.h` for the header file of the `foobar` module which defines API components.

- Use `foobardata.h` for the header file of the `foobar` module which defines interfaces used by the application configuration.
- Use `foobarimpl.h` for the header file of the `foobar` module which defines interfaces, macros, and inline functions used by the implementation.
- Do not place inline functions which are only used in one implementation source file into the implementation header file. Add these inline functions directly to the corresponding source file.
- Document all elements in header files with comments in Doxygen markup, see *Doxygen Guidelines* (page 157).
- Only place header files which should be directly included by the user with an `@file` Doxygen directive into the API documentation group. Place internal API header files with an `@file` Doxygen command into the implementation documentation group even if they define API elements. The API documentation group should only list public header files and no internal header files.

#### 6.3.1.13 Layering

- TBD: add something about the dependencies and header file layering.
- Understand the RTEMS Software Architecture.

#### 6.3.1.14 Tools

Some of the above can be assisted by tool support. Feel free to add more tools, configurations, etc here.

- `clang-format` – TODO.

### 6.3.2 Code Formatting

#### 6.3.2.1 Rules

- Minimize reformatting existing code in RTEMS unless the file undergoes substantial non-style changes.
- Adhere to the *Eighty Character Line Limit* (page 154).
- Use spaces instead of tabs.
- Use two spaces for one indentation level.
- Put function return types and names on one line if they fit.
- Put function calls on one line if they fit.
- No space between a function name or function-like macro and the opening parenthesis.
- Put braces on the same line as and one space after the conditional expression ends.
- Put the opening brace of a function definition one line after the closing parenthesis of its prototype.
- Put a single space inside and outside of each parenthesis of a conditional expression. Exception: never put a space before a closing semi-colon.
- Put a single space before and after ternary operators.

- Put a single space before and after binary operators.
- Put no space between unary operators (e.g. `*`, `&`, `!`, `~`, `++`, `--`) and their operands.
- No spaces around dereferencing operators (`->` and `.`).
- Do not use more than one blank line in a row.
- Do not use trailing white space at the end of a line.

#### 6.3.2.2 Eighty Character Line Limit

Code should look good for everyone under some standard width assumptions. Where a line wraps should be the same for anyone reading the code. For historical reasons, RTEMS uses 80 characters as the maximum width for each line of code. The newline (`\n`) character terminating the line does not count for the 80 character limit.

If you find yourself with code longer than 80 characters, first ask yourself whether the nesting level is too deep, names too long, compound expressions too complicated, or if some other guideline for improving readability can help to shrink the line length. Refactoring nested blocks into functions can help to alleviate code width problems while improving code readability. Making names descriptive yet terse can also improve readability. If absolutely necessary to have a long line, follow the rules on this page to break the line up to adhere to the 80 characters per line rule.

#### 6.3.2.3 Breaking Long Lines

The `if`, `while`, and `for` control statements have their condition expressions aligned and broken on separate lines. When the conditions have to be broken, none go on the first line with the `if`, `while`, or `for` statement, and none go on the last line with the closing parenthesis and the curly brace. Long statements are broken up and indented at operators, with an operator always being the last token on a line. No blank spaces should be left at the end of any line. The continuation of a broken line is indented by one level. Here is an example with a `for` loop.

```
for ( initialization = statement; a + really + longish + statement + that +␣
↪evaluates + to < a + boolean; another + statement ) {
  some_variable = a + really + longish + statement + that + needs + two + lines +␣
↪gets + indented + four + more + spaces + on + the + second + and + subsequent +␣
↪lines + and + broken + up + at + operators;
}
```

Should be replaced with

```
for (
  initialization = statement;
  a + really + longish + statement + that + evaluates + to <
    a + boolean;
  another + statement
) {
  some_variable = a + really + longish + statement + that + needs +
    two + lines + gets + indented + four + more +
    spaces + on + the + second + and + subsequent +
    lines + and + broken + up + at + operators;
}
```

Similarly,

```
if ( this + that < those && this + these < that && this + those < these && this <␣
↪those && those < that ) {
```

should be broken up like

```
if (
  this + that < those &&
  this + these < that &&
  this + those < these &&
  this < those &&
  those < that
) {
```

Note that each expression that resolves to a boolean goes on its own line. Where you place the boolean operator is a matter of choice.

When a line is long because of a comment at the end, move the comment to just before the line, for example

```
#define A_LONG_MACRO_NAME (AND + EXPANSION) /* Plus + a + really + long + comment␣
↪*/
```

can be replaced with

```
/* Plus + a + really + long + comment */
#define A_LONG_MACRO_NAME (AND + EXPANSION)
```

C Preprocessor macros need to be broken up with some care, because the preprocessor does not understand that it should eat newline characters. So

```
#define A_LONG_MACRO_NAME (AND + EXCESSIVELY + LONG + EXPANSION + WITH + LOTS +␣
↪OF + EXTRA + STUFF + DEFINED)
```

would become

```
#define A_LONG_MACRO_NAME ( \
  AND + EXCESSIVELY + LONG + EXPANSION + WITH + LOTS + OF + EXTRA + STUFF + \
  DEFINED \
)
```

Notice that each line is terminated by a backslash. The backslash tells the preprocessor to eat the newline. Of course, if you have such a long macro, you should consider not using a macro.

Function declarations can be broken up at each argument, for example

```
int this_is_a_function( int arg1, int arg2, int arg3, int arg4, int arg5, int␣
↪arg6, int arg7, int arg8, int arg9 );
```

would be broken up as

```
int this_is_a_function(
  int arg1,
  int arg2,
  int arg3,
  int arg4,
  int arg5,
  int arg6,
  int arg7,
  int arg8,
  int arg9
);
```

Excessively long comments should be broken up at a word boundary or somewhere that makes sense, for example

```
/* Excessively long comments should be broken up at a word boundary or somewhere␣
↪that makes sense, for example */
```

would be

```
/*
 * Excessively long comments should be broken up at a word boundary or
 * somewhere that makes sense, for example.
 */
```

Note that multiline comments have a single asterisk aligned with the asterisk in the opening `/*`. The closing `*/` should appear on a line by itself at the end.

### 6.3.3 Deprectating Interfaces

#### 6.3.3.1 Use the deprecate attribute

Add the `RTEMS_COMPILER_DEPRECATED_ATTRIBUTE`, which for gcc wraps the deprecated attribute, to functions, structures, and global symbols exported by the deprecated interface. Update the doxygen for each of these with the @deprecated command, for example:

```
/**
 * @brief RTEMS Feature
 *
 * @deprecated Feature is deprecated and will be removed.
 */
```

#### 6.3.3.2 Add a warning

Add a warning for configured features in confdefs.h

For features that are enabled or configured through confdefs.h, the feature should be disabled by default and a compile-time warning message should be printed, something along the lines of:

```
#warning "CONFIGURE_FEATURE_XXX\n\t\t\t**** Deprecated and will be removed. ****"
```

#### 6.3.3.3 Update documentation

Find references to the deprecated feature in the user manuals (doc) and wiki, and make a note that the features are deprecated and may be removed.

#### 6.3.3.4 Update support code

Update support code using deprecated feature

If there is support code using the feature, you will need to modify that support code to not use that feature. If the code cannot be immediately modified, file a ticket on the issue and disable the deprecated warning. The code will need to be addressed before the feature can be removed.

If the code in question is such that the feature's use can benignly be removed when the feature is removed, then simply disable the deprecated warning as shown below.

It is possible that a test may need to be split into two or more tests, so the code that is exercising the deprecated feature can be easily removed when the feature is removed.

#### 6.3.3.5 Disable deprecated warnings

After adding the deprecated attribute, the files which implement the method(s), any tests for them, and any support code using that feature that will remain until the feature is removed will need the deprecate warning disabled. If it is for an entire file, then using this:

```
/*
* We know this is deprecated and don't want a warning on every BSP built.
*/
#pragma GCC diagnostic ignored "-Wdeprecated-declarat
```

If it is for a section of code, then this is the appropriate code to surround the section with:

```
/*
* We know this is deprecated and don't want a warning on every BSP built.
*/
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"

/**** Code using deprecated feature ****/
#pragma GCC diagnostic pop
```

#### 6.3.3.6 Add a release note

Add the feature to a list of deprecated interfaces in the release notes.

### 6.3.4 Doxygen Guidelines

#### 6.3.4.1 Group Names

Doxygen group names shall use CamelCase. In the RTEMS source code, CamelCase is rarely used, so this makes it easier to search and replace Doxygen groups. It avoids ambiguous references to functions, types, defines, macros, and groups. All groups shall have an `RTEMS` prefix. This makes it possible to include the RTEMS files with Doxygen comments in a larger project without name conflicts. The group name shall use Title Case.

```
/**
 * @defgroup RTEMSScoreThread Thread Handler
 *
 * @ingroup RTEMSScore
 *
 * ...
 */
```

#### 6.3.4.2 Use Groups

Every file, function declaration, type definition, typedef, define, macro and global variable declaration shall belong to at least one Doxygen group. Use `@defgroup` and `@addtogroup` with `@{` and `@}` brackets to add members to a group. A group shall be defined at most once. Each group shall be documented with an `@brief` description and an optional detailed description. Use grammatically correct sentences for the `@brief` and detailed descriptions.

For the `@brief` description use phrases like this:

- This group contains ... and so on.
- The XYZ Handler provides ... and so on.
- The ABC Component contains ... and so on.

```
/**
 * @defgroup RTEMSScoreThread Thread Handler
 *
 * @ingroup RTEMSScore
 *
 * @brief The Thread Handler provides functionality related to the
 *   management of threads.
 *
 * This includes the creation, deletion, and scheduling of threads.
 *
 * ...
 *
 * @{
 */

... declarations, defines ...

/** @} */
```

```
/**
 * @addtogroup RTEMSScoreThread
 *
 * @{
 */

... declarations, defines ...

/** @} */
```

#### 6.3.4.3 Files

Each header and source file shall have an `@file` block at the top of the file after the SPDX License Identifier. The `@file` block shall precede the license header separated by one blank line, see *C/C++ Header File Template* (page 164) and *C/C++/Assembler Source File Template* (page 166). The `@file` block shall be put into a group with `@ingroup GroupName`. The `@file` block shall have an `@brief` description and an optional detailed description. The detailed description could give an explanation why a certain set of functions or data structures is grouped in one file. Use grammatically correct sentences for the `@brief` and detailed descriptions.

For the `@brief` description of header files use phrases like this:

- This header file provides ... and so on.
- This header file provides the API of the ABC Manager.
- This header file provides interfaces and functions used to implement the XYZ Handler.

For the `@brief` description of source files use phrases like this:

- This source file contains the implementation of some_function().
- This source file contains the definition of some_data_element.
- This source file contains the implementation of XZY Hander functions related to ABC processing.

```
/**
 * @file
 *
 * @ingroup RTEMSScoreThread
 *
 * @brief This source file contains the implementation of
 *   _Thread_Initialize().
 */
```

#### 6.3.4.4 Type Definitions

Each type (`typedef`, `struct`, `enum`) defined in a header file shall be documented with an `@brief` description and an optional detailed description. Use grammatically correct sentences for the `@brief` and detailed descriptions.

For the `@brief` description of types use phrases like this:

- This type represents ... and so on.
- This structure represents ... and so on.
- This structure provides ... and so on.
- This enumeration represents ... and so on.
- The XYZ represents ... and so on.

Each type member shall be documented with an `@brief` description and an optional detailed description. Use grammatically correct sentences for the `@brief` and detailed descriptions.

For the `@brief` description of types members use phrases like this:

- This member represents ... and so on.

- This member contains ... and so on.
- This member references ... and so on.
- The XYZ lock protects ... and so on.

For the `@brief` description of boolean type members use a phrase like this: "This member is true, if some condition is satisfied, otherwise it is false.".

```
/**
 * @brief The object information structure maintains the objects of an
 *   object class.
 *
 * If objects for the object class are configured, then an instance of this
 * structure is statically allocated and pre-initialized by
 * OBJECTS_INFORMATION_DEFINE() through <rtems/confdefs.h>.  The RTEMS
 * library contains a statically allocated and pre-initialized instance for
 * each object class providing zero objects, see
 * OBJECTS_INFORMATION_DEFINE_ZERO().
 */
typedef struct {
  /**
   * @brief This member contains the object identifier maximum of this
   *   object class.
   *
   * It is statically initialized.  The object identifier maximum provides
   * also the object API, class, and multiprocessing node information.
   *
   * It is used by _Objects_Get() to validate an object identifier.
   */
  Objects_Id maximum_id;

  ... more members ...
} Objects_Information;
```

#### 6.3.4.5 Function Declarations

Each function declaration or function-like macro in a header file shall be documented with an `@brief` description and an optional detailed description. Use grammatically correct sentences for the `@brief` and detailed descriptions. Use the descriptive-style for `@brief` descriptions, for example `"Creates a task."`, `"Sends the events to the task."`, or `"Obtains the semaphore."`. Use "the" to refer to parameters of the function. Do not use descriptions like `"Returns this and that."`. Describe the function return in `@retval` and `@return` paragraphs.

Each parameter shall be documented with an `@param` entry. List the `@param` entries in the order of the function parameters. For *non-const pointer* parameters

- use `@param[out]`, if the function writes under some conditions to memory locations referenced directly or indirectly by the non-`const` pointer parameter, or
- use `@param[in, out]`, if the function reads under some conditions from memory locations referenced directly or indirectly by the non-`const` pointer parameter and the function writes under some conditions to memory locations referenced directly or indirectly by the non-`const` pointer parameter.

If the function only reads from memory locations referenced directly or indirectly by a non-`const` pointer parameter, then the pointer parameter should be made `const`.

For other parameters (e.g. *const pointer* and *scalar* parameters) do not use the `[in]`, `[out]` or `[in, out]` parameter specifiers.

For the `@param` descriptions use phrases like this:

- is the ABC.
- indicates what should be done.
- defines the something.
- references the object to deal with.

The phrase shall form a grammatically correct sentence if "This parameter" precedes the phrase, for example "This parameter is the size of the message in bytes to send.".

Distinctive return values shall be documented with an `@retval` entry. Document the most common return value first. Use `@return` to describe the return of non-distinctive values. Use grammatically correct sentences for the descriptions. Use sentences in simple past tense to describe conditions which resulted in the return of a status value. Place `@retval` descriptions before the `@return` description. For functions returning a boolean value, use `@return` and a phrase like this: "Returns true, if some condition is satisfied, otherwise false.".

```
/**
 * @brief Sends a message to the message queue.
 *
 * This directive sends the message buffer to the message queue indicated by
 * ID.  If one or more tasks is blocked waiting to receive a message from this
 * message queue, then one will receive the message.  The task selected to
 * receive the message is based on the task queue discipline algorithm in use
 * by this particular message queue.  If no tasks are waiting, then the message
 * buffer will be placed at the rear of the chain of pending messages for this
 * message queue.
 *
 * @param id The message queue ID.
 * @param buffer The message content buffer.
 * @param size The size of the message.
 *
 * @retval RTEMS_SUCCESSFUL Successful operation.
 * @retval RTEMS_INVALID_ID Invalid message queue ID.
 * @retval RTEMS_INVALID_ADDRESS The message buffer pointer is @c NULL.
 * @retval RTEMS_INVALID_SIZE The message size is larger than the maximum
 *   message size of the message queue.
 * @retval RTEMS_TOO_MANY The new message would exceed the message queue limit
 *   for pending messages.
 */
rtems_status_code rtems_message_queue_send(
  rtems_id    id,
  const void *buffer,
  size_t      size
);
```

```
/**
 * @brief Receives a message from the message queue
 *
 * This directive is invoked when the calling task wishes to receive a message
 * from the message queue indicated by ID. The received message is to be placed
 * in the buffer. If no messages are outstanding and the option set indicates
 * that the task is willing to block, then the task will be blocked until a
 * message arrives or until, optionally, timeout clock ticks have passed.
 *
 * @param id The message queue ID.
 * @param[out] buffer The buffer for the message content.  The buffer must be
 *   large enough to store maximum size messages of this message queue.
 * @param[out] size The size of the message.
 * @param option_set The option set, e.g. RTEMS_NO_WAIT or RTEMS_WAIT.
 * @param timeout The number of ticks to wait if the RTEMS_WAIT is set.  Use
 *   RTEMS_NO_TIMEOUT to wait indefinitely.
 *
 * @retval RTEMS_SUCCESSFUL Successful operation.
 * @retval RTEMS_INVALID_ID Invalid message queue ID.
 * @retval RTEMS_INVALID_ADDRESS The message buffer pointer or the message size
 *   pointer is @c NULL.
 * @retval RTEMS_TIMEOUT A timeout occurred and no message was received.
 */
rtems_status_code rtems_message_queue_receive(
  rtems_id        id,
  void           *buffer,
  size_t         *size,
  rtems_option    option_set,
  rtems_interval  timeout
);
```

```
/**
 * @brief Allocates a memory block of the specified size from the workspace.
 *
 * @param size is the size in bytes of the memory block.
 *
 * @retval NULL No memory block with the requested size was available in the
 *   workspace.
 *
 * @return Returns the pointer to the allocated memory block, if enough
 *   memory to satisfy the allocation request was available.  The pointer is at
 *   least aligned by #CPU_HEAP_ALIGNMENT.
 */
void *_Workspace_Allocate( size_t size );
```

```
/**
 * @brief Rebalances the red-black tree after insertion of the node.
 *
 * @param[in, out] the_rbtree references the red-black tree.
```

(continues on next page)

(continued from previous page)

```
 * @param[in, out] the_node references the most recently inserted node.
 */
void _RBTree_Insert_color(
  RBTree_Control *the_rbtree,
  RBTree_Node    *the_node
);
```

```
/**
 * @brief Builds an object ID from its components.
 *
 * @param the_api is the object API.
 * @param the_class is the object class.
 * @param node is the object node.
 * @param index is the object index.
 *
 * @return Returns the object ID built from the specified components.
 */
#define _Objects_Build_id( the_api, the_class, node, index )
```

#### 6.3.4.6 Header File Examples

The <rtems/score/thread.h> and <rtems/score/threadimpl.h> header files are a good example of how header files should be documented.

### 6.3.5 File Templates

Every source code file shall have a copyright and license block. Corresponding to the license, every file shall have an SPDX License Identifier in the first possible line of the file. C/C++ files should have a Doxygen file comment block.

The preferred license for source code is BSD-2-Clause. The preferred license for documentation is CC-BY-SA-4.0.

#### 6.3.5.1 Copyright and License Block

You are the copyright holder. Use the following copyright and license block for your source code contributions to the RTEMS Project. Place it after the SPDX License Identifier line and the optional file documentation block.

- In case you are a real person, then use the following format for <COPYRIGHT HOLDER>: <FIRST NAME> <MIDDLE NAMES> <LAST NAME>. The <FIRST NAME> is your first name (also known as given name), the <MIDDLE NAMES> are your optional middle names, the <LAST NAME> is your last name (also known as family name).
- URLs are not permitted within the copyright block except for an email address for the copyright holder/author's contact information.
- Replace the <FIRST YEAR> placeholder with the year of your first substantial contribution to this file. Update the <LAST YEAR> with the year of your last substantial contribution to this file. If the first and last years are the same, then remove the <LAST YEAR> placeholder with the comma. Replace the <COPYRIGHT HOLDER> placeholder with your name.

- If more than one copyright holder exists for a file, then sort the copyright lines by the first year (earlier years are below later years) followed by the copyright holder in alphabetical order (A is above B in the file).

Use the following template for a copyright and license block. Do not change the license text.

```
Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>

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.
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.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Check the top-level `COPYING` file of the repository. If you are a new copyright holder, then add yourself to the top of the list. If your last year of a substantial contribution changed, then please update your copyright line.

#### 6.3.5.2 C/C++ Header File Template

Use the following guidelines and template for C and C++ header files (here `<foo/bar/baz.h>`):

- Place the SPDX License Identifier in the first line of the file.
- Add a Doxygen file documentation block.
- Place the copyright and license comment block after the documentation block.
- For the <FIRST YEAR>, <LAST YEAR>, and <COPYRIGHT HOLDER> placeholders see *Copyright and License Block* (page 163).
- Separate comment blocks by exactly one blank line.
- Separate the Doxygen comment block from the copyright and license, so that the copyright and license information is not included in the Doxygen output.
- For C++ header files discard the `extern "C"` start and end sections.

```
/* SPDX-License-Identifier: BSD-2-Clause */

```

(continues on next page)

(continued from previous page)

```
/**
 * @file
 *
 * @ingroup TheGroupForThisFile
 *
 * @brief Short "Table of Contents" Description of File Contents
 *
 * A short description of the purpose of this file.
 */

/*
 * Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>
 *
 * 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.
 * 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.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef _FOO_BAR_BAZ_H
#define _FOO_BAR_BAZ_H

#include <foo/bar/xyz.h>

/* Remove for C++ code */
#ifdef __cplusplus
extern "C" {
#endif

/* Declarations, defines, macros, inline functions, etc. */

/* Remove for C++ code */
#ifdef __cplusplus
```

(continues on next page)

(continued from previous page)

```
}
#endif

#endif /* _FOO_BAR_BAZ_H */
```

#### 6.3.5.3 C/C++/Assembler Source File Template

Use the following template for C, C++, and assembler source files (here implementation of <foo/bar/baz.h>). For the <FIRST YEAR>, <LAST YEAR>, and <COPYRIGHT HOLDER> placeholders see *Copyright and License Block* (page 163).

```
/* SPDX-License-Identifier: BSD-2-Clause */

/**
 * @file
 *
 * @ingroup TheGroupForThisFile
 *
 * @brief Short "Table of Contents" Description of File Contents
 *
 * A short description of the purpose of this file.
 */

/*
 * Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>
 *
 * 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.
 * 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.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifdef HAVE_CONFIG_H
```

(continues on next page)

(continued from previous page)

```
#include "config.h"
#endif

#include <foo/bar/baz.h>

/* Definitions, etc. */
```

#### 6.3.5.4 Python File Template

Use the following template for Python source files. For the <FIRST YEAR>, <LAST YEAR>, and <COPYRIGHT HOLDER> placeholders see *Copyright and License Block* (page 163).

The `File documentation block` is a Python docstring (PEP 257) module documentation block. RTEMS uses `"""` for Python docstrings.

```
# SPDX-License-Identifier: BSD-2-Clause
"""File documentation block"""

# Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>
#
# 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.
# 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.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
```

If the Python source file is a command line command add the following as the first line of the file:

```
#!/usr/bin/env python
```

A command line Python module does not need to have the `.py` file extension.

Only specify `python` as the command to `env`. A system that does not provide the `python` command can install a virtual python environment or the user can prepend the specific Python versioned command to the Python script on the command line when invoking the command.

#### 6.3.5.5 Shell Scripts

Use the following template for shell script source files and other files which accept a #-style comment block. For the <FIRST YEAR>, <LAST YEAR>, and <COPYRIGHT HOLDER> placeholders see *Copyright and License Block* (page 163).

```
#!/bin/sh
# SPDX-License-Identifier: BSD-2-Clause

# File documentation block

# Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>
#
# 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.
# 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.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
```

#### 6.3.5.6 reStructuredText File Template

Use the following template for reStructuredText (reST) source files. For the <FIRST YEAR>, <LAST YEAR>, and <COPYRIGHT HOLDER> placeholders see *Copyright and License Block* (page 163).

```
.. SPDX-License-Identifier: CC-BY-SA-4.0

.. Copyright (C) <FIRST YEAR>, <LAST YEAR> <COPYRIGHT HOLDER>
```

### 6.3.6 Naming Rules

#### 6.3.6.1 General Rules

- Avoid abbreviations.
  - Exception: When the abbreviation is more common than the full word.
  - Exception: For well-known acronyms.

- Use descriptive language.
- File names should be lower-case alphabet letters only, plus the extension. Avoid symbols in file names.
  - Exception: Use a single underscore _ or hyphen - to separate words in file names.
- Prefer to use underscores (Snake_Case) to separate words, rather than CamelCase or TitleCase.
- Local-scope variable names are all lower case with underscores between words.
- CPP macros are all capital letters with underscores between words.
- Enumerated (enum) values are all capital letters with underscores between words, but the type name follows the regular rules of other type names.
- Constant (const) variables follow the same rules as other variables. An exception is that a const that replaces a CPP macro might be all capital letters for backward compatibility.
- Type names, function names, and global scope names have different rules depending on whether they are part of the public API or are internal to RTEMS, see below.

#### 6.3.6.2 User-facing API

The public API routines follow a standard API like POSIX or BSD or start with `rtems_`. If a name starts with `rtems_`, then it should be assumed to be available for use by the application and be documented in the User's Guide.

The POSIX API follows the rules of POSIX.

#### 6.3.6.3 RTEMS internal interfaces

The SuperCore (`cpukit/score`) or "score" is organized in an object-oriented fashion. Each score Manager is a Package (or Module), and each Module contains type definitions, functions, etc. The following summarizes our conventions for using names within SuperCore Modules:

- Use `Module_name_Particular_type_name` for type names.
- Use `_Module_name_Particular_function_name` for function names.
- Use `_Module_name_Global_or_file_scope_variable_name` for global or file scope variable names.
- Within a structure:
  - Use `Name` for struct aggregate members.
  - Use `name` for reference members.
  - Use `name` for primitive type members.
  - Example:

```
typedef struct {
  Other_module_Struct_type    Aggregate_member_name;
  Other_module_Struct_type   *reference_member_name;
  Other_module_Primitive_type primitive_member_name;
} The_module_Type_name;
```

## 6.4 Documentation Guidelines

### 6.4.1 Application Configuration Options

Add at least an index entry and a label for the configuration option. Use a pattern of `CONFIGURE_[A-Z0-9_]+` for the option name. Use the following template for application configuration feature options:

```
.. index:: CONFIGURE_FEATURE

.. _CONFIGURE_FEATURE:

CONFIGURE_FEATURE
-----------------

CONSTANT:
    ``CONFIGURE_FEATURE``

OPTION TYPE:
    This configuration option is a boolean feature define.

DEFAULT CONFIGURATION:
    If this configuration option is undefined, then the described feature is not
    enabled.

DESCRIPTION:
    In case this configuration option is defined, then feature happens.

NOTES:
    Keep the description short.  Add all special cases, usage notes, error
    conditions, configuration dependencies, references, etc. here to the notes.
```

Use the following template for application configuration integer and initializer options:

```
.. index:: CONFIGURE_VALUE

.. _CONFIGURE_VALUE:

CONFIGURE_VALUE
---------------

CONSTANT:
    ``CONFIGURE_VALUE``

OPTION TYPE:
    This configuration option is an integer (or initializer) define.

DEFAULT VALUE:
    The default value is X.

VALUE CONSTRAINTS:
```

(continues on next page)

(continued from previous page)

```
    The value of this configuration option shall satisfy all of the following
    constraints:

    * It shall be greater than or equal to A.

    * It shall be less than or equal to B.

    * It shall meet C.

DESCRIPTION:
    The value of this configuration option defines the Y of Z in W.

NOTES:
    Keep the description short.  Add all special cases, usage notes, error
    conditions, configuration dependencies, references, etc. here to the notes.
```

## 6.5 Python Development Guidelines

Python is the preferred programming language for the RTEMS Tools. The RTEMS Tools run on the host computer of an RTEMS user or maintainer. These guidelines cover the Python language version, the source code formatting, use of static analysis tools, type annotations, testing, code coverage, and documentation. There are exceptions for existing code and third-party code. It is recommended to read the PEP 8 - Style Guide for Python Code and the Google Python Style Guide.

### 6.5.1 Python Language Versions

Although the official end-of-life of Python 2.7 was on January 1, 2020, the RTEMS Project still cares about Python 2.7 compatibility for some tools. Every tool provided by the RTEMS Project which an RTEMS user may use to develop applications with RTEMS should be Python 2.7 compatible. Examples are the build system, the RTEMS Source Builder, and the RTEMS Tester. The rationale is that there are still some maintained Linux distributions in the wild which ship only Python 2.7 by default. An example is CentOS 7 which gets maintenance updates until June 2024. Everything an RTEMS maintainer uses should be written in Python 3.6.

### 6.5.2 Python Code Formatting

Good looking code is important. Unfortunately, what looks good is a bit subjective and varies from developer to developer. Arguing about the code format is not productive. Code reviews should focus on more important topics, for example functionality, testability, and performance. Fortunately, for Python there are some good automatic code formatters available. All new code specifically developed for the RTEMS Tools should be piped through the yapf Python code formatter before it is committed or sent for review. Use the default settings of the tool (PEP 8 coding style).

You can disable the automatic formatting by the tool in a region starting with the `#yapf: disable` comment until the next `# yapf: enable` comment, for example

```
# yapf: disable
FOO = {
    # ... some very large, complex data literal.
}

BAR = [
    # ... another large data literal.
]
# yapf: enable
```

For a single literal, you can disable the formatting like this:

```
BAZ = {
    (1, 2, 3, 4),
    (5, 6, 7, 8),
    (9, 10, 11, 12),
}  # yapf: disable
```

### 6.5.3 Static Analysis Tools

Use the `flake8` and `pylint` static analysis tools for Python. Do not commit your code or send it for review if the tools find some rule violations. Run the tools with the default configuration. If you have problems to silence the tools, then please ask for help on the Developers Mailing List. Consult the tool documentation to silence false positives.

### 6.5.4 Type Annotations

For Python 3.6 or later code use type annotations. All public functions of your modules should have PEP 484 type annotations. Check for type issues with the mypy static type checker.

### 6.5.5 Testing

Write tests for your code with the pytest framework. Use the monkeypatch mocking module. Do not use the standard Python `unittest` and `unittest.mock` modules. Use `coverage run -m pytest` to run the tests with code coverage support. If you modify existing code or contribute new code to a subproject which uses tests and the code coverage metric, then do not make the code coverage worse.

#### 6.5.5.1 Test Organization

Do not use test classes to group tests. Use separate files instead. Avoid deep test directory hierarchies. For example, place tests for `mymodule.py` in `tests/test_mymodule.py`. For class-specific tests use:

- `mymodule.py:class First` → `tests/test_mymodule_first.py`
- `mymodule.py:class Second` → `tests/test_mymodule_second.py`
- `mymodule.py:class Third` → `tests/test_mymodule_third.py`

You can also group tests in other ways, for example:

- `mymodule.py` → `tests/test_mymodule_input.py`
- `mymodule.py` → `tests/test_mymodule_output.py`

### 6.5.6 Documentation

Document your code using the PEP 257 - Docstring Conventions. Contrary to PEP 257, use the descriptive-style (`"""Fetches rows from a Bigtable."""`) instead of imperative-style (`"""Fetch rows from a Bigtable."""`) as recommended by Comments and Docstrings - Functions and Methods. Use the Sphinx format. The sphinx-autodoc-typehints helps to reuse the type annotations for the documentation. Test code does not need docstrings in general.

### 6.5.7 Existing Code

Existing code in the RTEMS Tools may not follow the preceding guidelines. The RTEMS Project welcomes contributions which bring existing code in line with the guidelines. Firstly, run the `yapf` code formatter through the existing code of interest. Add `# yapf: disable` comments to avoid reformatting in some areas if it makes sense. If the existing code has no unit tests, then add unit tests before you modify existing code by hand. With the new unit tests aim at a good code coverage especially in the areas you intend to modify. While you review the code add docstrings. Run the static analysers and fix the rule violations. Please keep in mind that also trivial modifications can break working code. Make sure you have some unit tests. Add

type annotations unless the code should be Python 2.7 compatible. Concentrate on the public interfaces.

### 6.5.8 Third-Party Code

Try to not modify imported third-party code. In case there are issues with third-party code, then at least write a bug report or otherwise contact the upstream project. Reimport the third-party code after the issue is fixed in the upstream project. Only temporarily modify imported third-party code until a solution integrated in the upstream is available.

## 6.6 Change Management

Major decisions about RTEMS are made by the core developers in concert with the user community, guided by the Mission Statement. We provide access to our development sources via a Git Repository (see these Instructions for details).

TBD - ??? what in the Wiki could go here

## 6.7 Issue Tracking

The RTEMS Project uses Trac to manage all change requests and problem reports and refers to either as a ticket.

The bug reporting procedure is documented in the RTEMS User Manual.

TBD Review process, workflows, etc.

CHAPTER

**SEVEN**

# SOFTWARE TEST PLAN ASSURANCE AND PROCEDURES

## 7.1 Testing and Coverage

Testing to verify that requirements are implemented is a critical part of the high integrity processes. Similarly, measuring and reporting source and decision path coverage of source code is critical.

Needed improvements to the RTEMS testing infrastructure should be done as part of the open project. Similarly, improvements in RTEMS coverage reporting should be done as part of the open project. Both of these capabilities are part of the RTEMS Tester toolset.

Assuming that a requirements focused test suite is added to the open RTEMS, tools will be needed to assist in verifying that requirements are "fully tested." A fully tested requirement is one which is implemented and tested with associated logical tracing. Tools automating this analysis and generating reporting and alerts will be a critical part of ensuring the source technical data does not bit rot.

Must use tools from:

RTEMS Tools Project: https://gitlab.rtems.org/rtems/tools/rtems-tools

Scope, Procedures, Methodologies, Tools TBD - Write content

### 7.1.1 Test Suites

All RTEMS source distributions include the complete RTEMS test suites. These tests must be compiled and linked for a specific BSP. Some BSPs are for freely available simulators and thus anyone may test RTEMS on a simulator. Most of the BSPs which can execute on a simulator include scripts to help automate running them.

The RTEMS Project welcomes additions to the various test suites and sample application collections. This helps improve coverage of functionality as well as ensure user use cases are regularly tested.

The following functional test suites are included with RTEMS.

- Classic API Single Processor Test Suite
- POSIX API Test Suite
- File System Test Suite
- Support Library Test Suite (libtests)
- Symmetric Multiprocessing Test Suite
- Distributed Multiprocessing Test Suite
- Classic API Ada95 Binding Test Suite

The following timing test suites are included with RTEMS.

- Classic API Timing Test Suite
- POSIX API Timing Test Suite
- Rhealstone Collection
- Benchmarks Collecction

The RTEMS source distribution includes two collections of sample applications.

- Sample Applications (built as RTEMS tests)

- Example Applications (built as RTEMS user applications)

The RTEMS libbsd package includes its own test suite.

#### 7.1.1.1 Legacy Test Suites

The following are available for the legacy IPV4 Network Stack:

- Network Demonstration Applications

Post RTEMS 4.10, ITRON API support was removed. The following test suites are only available if the ITRON API support is present in RTEMS.

- ITRON API Test Suite
- ITRON API Timing Test Suite

### 7.1.2 RTEMS Tester

The RTEMS Tester is a test tool which provides a command line interface and automates execution of test executables. It is part of the `rtems-tools` repository and built as part of the RTEMS Tools for all targets by the RTEMS Source Builder. The RTEMS Tester can be configured to test based on local lab setup or to test on custom boards.

The RTEMS Tester is documented the `RTEMS Tester and Run` section of the `RTEMS User Manual`.

CHAPTER

**EIGHT**

# SOFTWARE TEST FRAMEWORK

## 8.1 The RTEMS Test Framework

The `RTEMS Test Framework` helps you to write test suites. It has the following features:

- Implemented in standard C11
- Tests can be written in C or C++
- Runs on at least FreeBSD, MSYS2, Linux and RTEMS
- Test runner and test case code can be in separate translation units
- Test cases are automatically registered at link-time
- Test cases may have a test fixture
- Test checks for various standard types
- Supports test case planning
- Test case scoped dynamic memory
- Test case destructors
- Test case resource accounting to show that no resources are leaked during the test case execution
- Supports early test case exit, e.g. in case a malloc() fails
- Individual test case and overall test suite duration is reported
- Procedures for code runtime measurements in RTEMS
- Easy to parse test report to generate for example human readable test reports
- Low overhead time measurement of short time sequences (using cycle counter hardware if a available)
- Configurable time service provider for a monotonic clock
- Low global memory overhead for test cases and test checks
- Supports multi-threaded execution and interrupts in test cases
- A simple (polled) put character function is sufficient to produce the test report
- Only text, global data and a stack pointer must be set up to run a test suite
- No dynamic memory is used by the framework itself
- No memory is aggregated throughout the test case execution

### 8.1.1 Nomenclature

A `test suite` is a collection of test cases. A `test case` consists of individual test actions and checks. A `test check` determines if the outcome of a test action meets its expectation. A `test action` is a program sequence with an observable outcome, for example a function invocation with a return status. If a test action produces the expected outcome as determined by the corresponding test check, then this test check passes, otherwise this test check fails. The test check failures of a test case are summed up. A test case passes, if the failure count of this test case is zero, otherwise the test case fails. The test suite passes if all test cases pass, otherwise it fails.

### 8.1.2 Test Cases

You can write a test case with the `T_TEST_CASE()` macro followed by a function body:

```
T_TEST_CASE(name)
{
   /* Your test case code */
}
```

The test case `name` must be a valid C designator. The test case names must be unique within the test suite. Just link modules with test cases to the test runner to form a test suite. The test cases are automatically registered via static C constructors.

Listing 8.1: Test Case Example

```
#include <t.h>

static int add(int a, int b)
{
    return a + b;
}

T_TEST_CASE(a_test_case)
{
    int actual_value;

    actual_value = add(1, 1);
    T_eq_int(actual_value, 2);
    T_true(false, "a test failure message");
}
```

Listing 8.2: Test Case Report

```
B:a_test_case
P:0:8:UI1:test-simple.c:13
F:1:8:UI1:test-simple.c:14:a test failure message
E:a_test_case:N:2:F:1:D:0.001657
```

The `B` line indicates the begin of test case `a_test_case`. The `P` line shows that the test check in file `test-simple.c` at line 13 executed by task `UI1` on processor 0 as the test step 0 passed. The invocation of `add()` in line 12 is the test action of test step 0. The `F` lines shows that the test check in file `test-simple.c` at line 14 executed by task `UI1` on processor 0 as the test step 1 failed with a message of `"a test failure message"`. The `E` line indicates the end of test case `a_test_case` resulting in a total of two test steps (`N`) and one test failure (`F`). The test case execution duration (`D`) was 0.001657 seconds. For test report details see: *Test Reporting* (page 211).

### 8.1.3 Test Fixture

You can write a test case with a test fixture with the `T_TEST_CASE_FIXTURE()` macro followed by a function body:

```
T_TEST_CASE_FIXTURE(name, fixture)
{
   /* Your test case code */
}
```

The test case `name` must be a valid C designator. The test case names must be unique within the test suite. The `fixture` must point to a statically initialized read-only object of type `T_fixture`.

```
typedef struct T_fixture {
    void (*setup)(void *context);
    void (*stop)(void *context);
    void (*teardown)(void *context);
    void (*scope)(void *context, char *buffer, size_t size);
    void *initial_context;
} T_fixture;
```

The test fixture provides methods to setup, stop, and teardown a test case as well as the scope for log messages. A context is passed to each of the methods. The initial context is defined by the read-only fixture object. The context can be obtained by the `T_fixture_context()` function. The context can be changed within the scope of one test case by the `T_set_fixture_context()` function. The next test case execution using the same fixture will start again with the initial context defined by the read-only fixture object. Setting the context can be used for example to dynamically allocate a test environment in the setup method.

The test case fixtures of a test case are organized as a stack. Fixtures can be dynamically added to a test case and removed from a test case via the `T_push_fixture()` and `T_pop_fixture()` functions.

```
void *T_push_fixture(T_fixture_node *node, const T_fixture *fixture);

void T_pop_fixture(void);
```

The `T_push_fixture()` function needs an uninitialized fixture node which must exist until `T_pop_fixture()` is called. It returns the initial context of the fixture. At the end of a test case all pushed fixtures are popped automatically. A call of `T_pop_fixture()` invokes the teardown method of the fixture and must correspond to a previous call to `T_push_fixture()`.

Listing 8.3: Test Fixture Example

```
#include <t.h>

static int initial_value = 3;

static int counter;

static void
setup(void *ctx)
{
    int *c;

    T_log(T_QUIET, "setup begin");
```

(continues on next page)

(continued from previous page)

```
    T_eq_ptr(ctx, &initial_value);
    T_eq_ptr(ctx, T_fixture_context());
    c = ctx;
    counter = *c;
    T_set_fixture_context(&counter);
    T_eq_ptr(&counter, T_fixture_context());
    T_log(T_QUIET, "setup end");
}

static void
stop(void *ctx)
{
    int *c;

    T_log(T_QUIET, "stop begin");
    T_eq_ptr(ctx, &counter);
    c = ctx;
    ++(*c);
    T_log(T_QUIET, "stop end");
}

static void
teardown(void *ctx)
{
    int *c;

    T_log(T_QUIET, "teardown begin");
    T_eq_ptr(ctx, &counter);
    c = ctx;
    T_eq_int(*c, 4);
    T_log(T_QUIET, "teardown end");
}

static const T_fixture fixture = {
    .setup = setup,
    .stop = stop,
    .teardown = teardown,
    .initial_context = &initial_value
};

T_TEST_CASE_FIXTURE(fixture, &fixture)
{
    T_assert_true(true, "all right");
    T_assert_true(false, "test fails and we stop the test case");
    T_log(T_QUIET, "not reached");
}
```

Listing 8.4: Test Fixture Report

```
B:fixture
L:setup begin
P:0:0:UI1:test-fixture.c:13
P:1:0:UI1:test-fixture.c:14
P:2:0:UI1:test-fixture.c:18
L:setup end
P:3:0:UI1:test-fixture.c:55
F:4:0:UI1:test-fixture.c:56:test fails and we stop the test case
L:stop begin
P:5:0:UI1:test-fixture.c:28
L:stop end
L:teardown begin
P:6:0:UI1:test-fixture.c:40
P:7:0:UI1:test-fixture.c:42
L:teardown end
E:fixture:N:8:F:1
```

### 8.1.4 Test Case Planning

A non-quiet test check fetches and increments the test step counter atomically. For each test case execution the planned steps can be specified with the `T_plan()` function.

```
void T_plan(unsigned int planned_steps);
```

This function must be invoked at most once in each test case execution. If the planned test steps are set with this function, then the final test steps after the test case execution must be equal to the planned steps, otherwise the test case fails.

Use the `T_step_*(step, ...)` test check variants to ensure that the test case execution follows exactly the planned steps.

Listing 8.5: Test Planning Example

```
#include <t.h>

T_TEST_CASE(wrong_step)
{
    T_plan(2);
    T_step_true(0, true, "all right");
    T_step_true(2, true, "wrong step");
}

T_TEST_CASE(plan_ok)
{
    T_plan(1);
    T_step_true(0, true, "all right");
}

T_TEST_CASE(plan_failed)
```

(continues on next page)

(continued from previous page)

```
{
    T_plan(2);
    T_step_true(0, true, "not enough steps");
    T_quiet_true(true, "quiet test do not count");
}

T_TEST_CASE(double_plan)
{
    T_plan(99);
    T_plan(2);
}

T_TEST_CASE(steps)
{
    T_step(0, "a");
    T_plan(3);
    T_step(1, "b");
    T_step(2, "c");
}
```

Listing 8.6: Test Planning Report

```
B:wrong_step
P:0:0:UI1:test-plan.c:6
F:1:0:UI1:test-plan.c:7:planned step (2)
E:wrong_step:N:2:F:1
B:plan_ok
P:0:0:UI1:test-plan.c:13
E:plan_ok:N:1:F:0
B:plan_failed
P:0:0:UI1:test-plan.c:19
F:*:0:UI1:*:*:actual steps (1), planned steps (2)
E:plan_failed:N:1:F:1
B:double_plan
F:*:0:UI1:*:*:planned steps (99) already set
E:double_plan:N:0:F:1
B:steps
P:0:0:UI1:test-plan.c:31
P:1:0:UI1:test-plan.c:33
P:2:0:UI1:test-plan.c:34
E:steps:N:3:F:0
```

### 8.1.5 Test Case Resource Accounting

The framework can check if various resources have leaked during a test case execution. The resource checkers are specified by the test run configuration. On RTEMS, checks for the following resources are available

- workspace and heap memory,
- file descriptors,

- POSIX keys and key value pairs,
- RTEMS barriers,
- RTEMS user extensions,
- RTEMS message queues,
- RTEMS partitions,
- RTEMS periods,
- RTEMS regions,
- RTEMS semaphores,
- RTEMS tasks, and
- RTEMS timers.

Listing 8.7: Resource Accounting Example

```
#include <t.h>

#include <stdlib.h>

#include <rtems.h>

T_TEST_CASE(missing_sema_delete)
{
    rtems_status_code sc;
    rtems_id id;

    sc = rtems_semaphore_create(rtems_build_name('S', 'E', 'M', 'A'), 0,
        RTEMS_COUNTING_SEMAPHORE, 0, &id);
    T_rsc_success(sc);
}

T_TEST_CASE(missing_free)
{
    void *p;

    p = malloc(1);
    T_not_null(p);
}
```

Listing 8.8: Resource Accounting Report

```
B:missing_sema_delete
P:0:0:UI1:test-leak.c:14
F:*:0:UI1:*:*:RTEMS semaphore leak (1)
E:missing_sema_delete:N:1:F:1:D:0.004013
B:missing_free
P:0:0:UI1:test-leak.c:22
F:*:0:UI1:*:*:memory leak in workspace or heap
E:missing_free:N:1:F:1:D:0.003944
```

### 8.1.6 Test Case Scoped Dynamic Memory

You can allocate dynamic memory which is automatically freed after the current test case execution. You can provide an optional destroy function to `T_zalloc()` which is called right before the memory is freed. The `T_zalloc()` function initializes the memory to zero.

```
void *T_malloc(size_t size);

void *T_calloc(size_t nelem, size_t elsize);

void *T_zalloc(size_t size, void (*destroy)(void *));

void T_free(void *ptr);
```

Listing 8.9: Test Case Scoped Dynamic Memory Example

```
#include <t.h>

T_TEST_CASE(malloc_free)
{
    void *p;

    p = T_malloc(1);
    T_assert_not_null(p);
    T_free(p);
}

T_TEST_CASE(malloc_auto)
{
    void *p;

    p = T_malloc(1);
    T_assert_not_null(p);
}

static void
destroy(void *p)
{
    int *i;

    i = p;
    T_step_eq_int(2, *i, 1);
}

T_TEST_CASE(zalloc_auto)
{
    int *i;

    T_plan(3);
    i = T_zalloc(sizeof(*i), destroy);
    T_step_assert_not_null(0, i);
```

(continues on next page)

(continued from previous page)

```
    T_step_eq_int(1, *i, 0);
    *i = 1;
}
```

Listing 8.10: Test Case Scoped Dynamic Memory Report

```
B:malloc_free
P:0:0:UI1:test-malloc.c:8
E:malloc_free:N:1:F:0:D:0.005200
B:malloc_auto
P:0:0:UI1:test-malloc.c:17
E:malloc_auto:N:1:F:0:D:0.004790
B:zalloc_auto
P:0:0:UI1:test-malloc.c:35
P:1:0:UI1:test-malloc.c:36
P:2:0:UI1:test-malloc.c:26
E:zalloc_auto:N:3:F:0:D:0.006583
```

### 8.1.7 Test Case Destructors

You can add test case destructors with `T_add_destructor()`. The destructors are called automatically at the test case end before the resource accounting takes place. Optionally, a registered destructor can be removed before the test case end with `T_remove_destructor()`. The `T_destructor` structure of a destructor must exist after the return from the test case body. It is recommended to use statically allocated memory. Do not use stack memory or dynamic memory obtained via `T_malloc()`, `T_calloc()` or `T_zalloc()` for the `T_destructor` structure.

```
void T_add_destructor(T_destructor *destructor,
   void (*destroy)(T_destructor *));

void T_remove_destructor(T_destructor *destructor);
```

Listing 8.11: Test Case Destructor Example

```
#include <t.h>

static void
destroy(T_destructor *dtor)
{
    (void)dtor;
    T_step(0, "destroy");
}

T_TEST_CASE(destructor)
{
    static T_destructor dtor;

    T_plan(1);
    T_add_destructor(&dtor, destroy);
}
```

Listing 8.12: Test Case Destructor Report

```
B:destructor
P:0:0:UI1:test-destructor.c:7
E:destructor:N:1:F:0:D:0.003714
```

### 8.1.8 Test Checks

A `test check` determines if the actual value presented to the test check has the expected properties. The actual value should represent the outcome of a test action. If a test action produces the expected outcome as determined by the corresponding test check, then this test check passes, otherwise this test check fails. A failed test check does not stop the test case execution immediately unless the `T_assert_*()` test variant is used. Each test check increments the test step counter unless the `T_quiet_*()` test variant is used. The test step counter is initialized to zero before the test case begins to execute. The `T_step_*(step, ...)` test check variants verify that the test step counter is equal to the planned test step value, otherwise the test check fails.

#### 8.1.8.1 Test Check Variant Conventions

The `T_quiet_*()` test check variants do not increment the test step counter and only print a message if the test check fails. This is helpful in case a test check appears in a tight loop.

The `T_step_*(step, ...)` test check variants check in addition that the test step counter is equal to the specified test step value, otherwise the test check fails.

The `T_assert_*()` and `T_step_assert_*(step, ...)` test check variants stop the current test case execution if the test check fails.

#### 8.1.8.2 Test Check Parameter Conventions

The following names for test check parameters are used throughout the test checks:

**step**
: The planned test step for this test check.

**a**
: The actual value to check against an expected value. It is usually the first parameter in all test checks, except in the `T_step_*(step, ...)` test check variants, here it is the second parameter.

**e**
: The expected value of a test check. This parameter is optional. Some test checks have an implicit expected value. If present, then this parameter is directly after the actual value parameter of the test check.

**fmt**
: A printf()-like format string. Floating-point and exotic formats may be not supported.

#### 8.1.8.3 Test Check Condition Conventions

The following names for test check conditions are used:

**eq**
: The actual value must equal the expected value.

**ne**
The actual value must not equal the value of the second parameter.

**ge**
The actual value must be greater than or equal to the expected value.

**gt**
The actual value must be greater than the expected value.

**le**
The actual value must be less than or equal to the expected value.

**lt**
The actual value must be less than the expected value.

If the actual value satisfies the test check condition, then the test check passes, otherwise it fails.

#### 8.1.8.4 Test Check Type Conventions

The following names for test check types are used:

**ptr**
The test value must be a pointer (`void *`).

**mem**
The test value must be a memory area with a specified length.

**str**
The test value must be a null byte terminated string.

**nstr**
The length of the test value string is limited to a specified maximum.

**char**
The test value must be a character (`char`).

**schar**
The test value must be a signed character (`signed char`).

**uchar**
The test value must be an unsigned character (`unsigned char`).

**short**
The test value must be a short integer (`short`).

**ushort**
The test value must be an unsigned short integer (`unsigned short`).

**int**
The test value must be an integer (`int`).

**uint**
The test value must be an unsigned integer (`unsigned int`).

**long**
The test value must be a long integer (`long`).

**ulong**
The test value must be an unsigned long integer (`unsigned long`).

**ll**
The test value must be a long long integer (`long long`).

**ull**
The test value must be an unsigned long long integer (`unsigned long long`).

**i8**
The test value must be a signed 8-bit integer (`int8_t`).

**u8**
The test value must be an unsigned 8-bit integer (`uint8_t`).

**i16**
The test value must be a signed 16-bit integer (`int16_t`).

**u16**
The test value must be an unsigned 16-bit integer (`uint16_t`).

**i32**
The test value must be a signed 32-bit integer (`int32_t`).

**u32**
The test value must be an unsigned 32-bit integer (`uint32_t`).

**i64**
The test value must be a signed 64-bit integer (`int64_t`).

**u64**
The test value must be an unsigned 64-bit integer (`uint64_t`).

**iptr**
The test value must be of type `intptr_t`.

**uptr**
The test value must be of type `uintptr_t`.

**ssz**
The test value must be of type `ssize_t`.

**sz**
The test value must be of type `size_t`.

#### 8.1.8.5 Integers

Let `xyz` be the type variant which shall be one of `schar`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `ll`, `ull`, `i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `iptr`, `uptr`, `ssz`, and `sz`.

Let `I` be the type name which shall be compatible to the type variant.

The following test checks for integers are available:

```
1 void T_eq_xyz(I a, I e);
2 void T_assert_eq_xyz(I a, I e);
3 void T_quiet_eq_xyz(I a, I e);
4 void T_step_eq_xyz(unsigned int step, I a, I e);
5 void T_step_assert_eq_xyz(unsigned int step, I a, I e);
6
7 void T_ne_xyz(I a, I e);
8 void T_assert_ne_xyz(I a, I e);
```

(continues on next page)

(continued from previous page)

```
void T_quiet_ne_xyz(I a, I e);
void T_step_ne_xyz(unsigned int step, I a, I e);
void T_step_assert_ne_xyz(unsigned int step, I a, I e);

void T_ge_xyz(I a, I e);
void T_assert_ge_xyz(I a, I e);
void T_quiet_ge_xyz(I a, I e);
void T_step_ge_xyz(unsigned int step, I a, I e);
void T_step_assert_ge_xyz(unsigned int step, I a, I e);

void T_gt_xyz(I a, I e);
void T_assert_gt_xyz(I a, I e);
void T_quiet_gt_xyz(I a, I e);
void T_step_gt_xyz(unsigned int step, I a, I e);
void T_step_assert_gt_xyz(unsigned int step, I a, I e);

void T_le_xyz(I a, I e);
void T_assert_le_xyz(I a, I e);
void T_quiet_le_xyz(I a, I e);
void T_step_le_xyz(unsigned int step, I a, I e);
void T_step_assert_le_xyz(unsigned int step, I a, I e);

void T_lt_xyz(I a, I e);
void T_assert_lt_xyz(I a, I e);
void T_quiet_lt_xyz(I a, I e);
void T_step_lt_xyz(unsigned int step, I a, I e);
void T_step_assert_lt_xyz(unsigned int step, I a, I e);
```

An automatically generated message is printed in case the test check fails.

#### 8.1.8.6 Boolean Expressions

The following test checks for boolean expressions are available:

```
void T_true(bool a, const char *fmt, ...);
void T_assert_true(bool a, const char *fmt, ...);
void T_quiet_true(bool a, const char *fmt, ...);
void T_step_true(unsigned int step, bool a, const char *fmt, ...);
void T_step_assert_true(unsigned int step, bool a, const char *fmt, ...);

void T_false(bool a, const char *fmt, ...);
void T_assert_false(bool a, const char *fmt, ...);
void T_quiet_false(bool a, const char *fmt, ...);
void T_step_false(unsigned int step, bool a, const char *fmt, ...);
void T_step_assert_false(unsigned int step, bool a, const char *fmt, ...);
```

The message is only printed in case the test check fails. The format parameter is mandatory.

Listing 8.13: Boolean Test Checks Example

```
#include <t.h>

T_TEST_CASE(example)
{
    T_true(true, "test passes, no message output");
    T_true(false, "test fails");
    T_quiet_true(true, "quiet test passes, no output at all");
    T_quiet_true(false, "quiet test fails");
    T_step_true(2, true, "step test passes, no message output");
    T_step_true(3, false, "step test fails");
    T_assert_false(true, "this is a format %s", "string");
}
```

Listing 8.14: Boolean Test Checks Report

```
B:example
P:0:0:UI1:test-example.c:5
F:1:0:UI1:test-example.c:6:test fails
F:*:0:UI1:test-example.c:8:quiet test fails
P:2:0:UI1:test-example.c:9
F:3:0:UI1:test-example.c:10:step test fails
F:4:0:UI1:test-example.c:11:this is a format string
E:example:N:5:F:4
```

#### 8.1.8.7 Generic Types

The following test checks for data types with an equality (==) or inequality (!=) operator are available:

```
void T_eq(T a, T e, const char *fmt, ...);
void T_assert_eq(T a, T e, const char *fmt, ...);
void T_quiet_eq(T a, T e, const char *fmt, ...);
void T_step_eq(unsigned int step, T a, T e, const char *fmt, ...);
void T_step_assert_eq(unsigned int step, T a, T e, const char *fmt, ...);

void T_ne(T a, T e, const char *fmt, ...);
void T_assert_ne(T a, T e, const char *fmt, ...);
void T_quiet_ne(T a, T e, const char *fmt, ...);
void T_step_ne(unsigned int step, T a, T e, const char *fmt, ...);
void T_step_assert_ne(unsigned int step, T a, T e, const char *fmt, ...);
```

The type name `T` specifies an arbitrary type which must support the corresponding operator. The message is only printed in case the test check fails. The format parameter is mandatory.

#### 8.1.8.8 Pointers

The following test checks for pointers are available:

```
void T_eq_ptr(const void *a, const void *e);
void T_assert_eq_ptr(const void *a, const void *e);
void T_quiet_eq_ptr(const void *a, const void *e);
void T_step_eq_ptr(unsigned int step, const void *a, const void *e);
void T_step_assert_eq_ptr(unsigned int step, const void *a, const void *e);

void T_ne_ptr(const void *a, const void *e);
void T_assert_ne_ptr(const void *a, const void *e);
void T_quiet_ne_ptr(const void *a, const void *e);
void T_step_ne_ptr(unsigned int step, const void *a, const void *e);
void T_step_assert_ne_ptr(unsigned int step, const void *a, const void *e);

void T_null(const void *a);
void T_assert_null(const void *a);
void T_quiet_null(const void *a);
void T_step_null(unsigned int step, const void *a);
void T_step_assert_null(unsigned int step, const void *a);

void T_not_null(const void *a);
void T_assert_not_null(const void *a);
void T_quiet_not_null(const void *a);
void T_step_not_null(unsigned int step, const void *a);
void T_step_assert_not_null(unsigned int step, const void *a);
```

An automatically generated message is printed in case the test check fails.

#### 8.1.8.9 Memory Areas

The following test checks for memory areas are available:

```
void T_eq_mem(const void *a, const void *e, size_t n);
void T_assert_eq_mem(const void *a, const void *e, size_t n);
void T_quiet_eq_mem(const void *a, const void *e, size_t n);
void T_step_eq_mem(unsigned int step, const void *a, const void *e, size_t n);
void T_step_assert_eq_mem(unsigned int step, const void *a, const void *e, size_t␣
↪n);

void T_ne_mem(const void *a, const void *e, size_t n);
void T_assert_ne_mem(const void *a, const void *e, size_t n);
void T_quiet_ne_mem(const void *a, const void *e, size_t n);
void T_step_ne_mem(unsigned int step, const void *a, const void *e, size_t n);
void T_step_assert_ne_mem(unsigned int step, const void *a, const void *e, size_t␣
↪n);
```

The `memcmp()` function is used to compare the memory areas. An automatically generated message is printed in case the test check fails.

#### 8.1.8.10 Strings

The following test checks for strings are available:

```
void T_eq_str(const char *a, const char *e);
void T_assert_eq_str(const char *a, const char *e);
void T_quiet_eq_str(const char *a, const char *e);
void T_step_eq_str(unsigned int step, const char *a, const char *e);
void T_step_assert_eq_str(unsigned int step, const char *a, const char *e);

void T_ne_str(const char *a, const char *e);
void T_assert_ne_str(const char *a, const char *e);
void T_quiet_ne_str(const char *a, const char *e);
void T_step_ne_str(unsigned int step, const char *a, const char *e);
void T_step_assert_ne_str(unsigned int step, const char *a, const char *e);

void T_eq_nstr(const char *a, const char *e, size_t n);
void T_assert_eq_nstr(const char *a, const char *e, size_t n);
void T_quiet_eq_nstr(const char *a, const char *e, size_t n);
void T_step_eq_nstr(unsigned int step, const char *a, const char *e, size_t n);
void T_step_assert_eq_nstr(unsigned int step, const char *a, const char *e, size_
↪t n);

void T_ne_nstr(const char *a, const char *e, size_t n);
void T_assert_ne_nstr(const char *a, const char *e, size_t n);
void T_quiet_ne_nstr(const char *a, const char *e, size_t n);
void T_step_ne_nstr(unsigned int step, const char *a, const char *e, size_t n);
void T_step_assert_ne_nstr(unsigned int step, const char *a, const char *e, size_
↪t n);
```

The `strcmp()` and `strncmp()` functions are used to compare the strings. An automatically generated message is printed in case the test check fails.

#### 8.1.8.11 Characters

The following test checks for characters (`char`) are available:

```
void T_eq_char(char a, char e);
void T_assert_eq_char(char a, char e);
void T_quiet_eq_char(char a, char e);
void T_step_eq_char(unsigned int step, char a, char e);
void T_step_assert_eq_char(unsigned int step, char a, char e);

void T_ne_char(char a, char e);
void T_assert_ne_char(char a, char e);
void T_quiet_ne_char(char a, char e);
void T_step_ne_char(unsigned int step, char a, char e);
void T_step_assert_ne_char(unsigned int step, char a, char e);
```

An automatically generated message is printed in case the test check fails.

#### 8.1.8.12 RTEMS Status Codes

The following test checks for RTEMS status codes are available:

```
void T_rsc(rtems_status_code a, rtems_status_code e);
void T_assert_rsc(rtems_status_code a, rtems_status_code e);
void T_quiet_rsc(rtems_status_code a, rtems_status_code e);
void T_step_rsc(unsigned int step, rtems_status_code a, rtems_status_code e);
void T_step_assert_rsc(unsigned int step, rtems_status_code a, rtems_status_code
↪e);

void T_rsc_success(rtems_status_code a);
void T_assert_rsc_success(rtems_status_code a);
void T_quiet_rsc_success(rtems_status_code a);
void T_step_rsc_success(unsigned int step, rtems_status_code a);
void T_step_assert_rsc_success(unsigned int step, rtems_status_code a);
```

An automatically generated message is printed in case the test check fails.

#### 8.1.8.13 POSIX Error Numbers

The following test checks for POSIX error numbers are available:

```
void T_eno(int a, int e);
void T_assert_eno(int a, int e);
void T_quiet_eno(int a, int e);
void T_step_eno(unsigned int step, int a, int e);
void T_step_assert_eno(unsigned int step, int a, int e);

void T_eno_success(int a);
void T_assert_eno_success(int a);
void T_quiet_eno_success(int a);
void T_step_eno_success(unsigned int step, int a);
void T_step_assert_eno_success(unsigned int step, int a);
```

The actual and expected value must be a POSIX error number, e.g. EINVAL, ENOMEM, etc. An automatically generated message is printed in case the test check fails.

#### 8.1.8.14 POSIX Status Codes

The following test checks for POSIX status codes are available:

```
void T_psx_error(int a, int eno);
void T_assert_psx_error(int a, int eno);
void T_quiet_psx_error(int a, int eno);
void T_step_psx_error(unsigned int step, int a, int eno);
void T_step_assert_psx_error(unsigned int step, int a, int eno);

void T_psx_success(int a);
void T_assert_psx_success(int a);
void T_quiet_psx_success(int a);
void T_step_psx_success(unsigned int step, int a);
void T_step_assert_psx_success(unsigned int step, int a);
```

The eno value must be a POSIX error number, e.g. EINVAL, ENOMEM, etc. An actual value of zero indicates success. An actual value of minus one indicates an error. An automatically generated message is printed in case the test check fails.

Listing 8.15: POSIX Status Code Example

```
#include <t.h>

#include <sys/stat.h>
#include <errno.h>

T_TEST_CASE(stat)
{
    struct stat st;
    int status;

    errno = 0;
    status = stat("foobar", &st);
    T_psx_error(status, ENOENT);
}
```

Listing 8.16: POSIX Status Code Report

```
B:stat
P:0:0:UI1:test-psx.c:13
E:stat:N:1:F:0
```

### 8.1.9 Log Messages and Formatted Output

You can print log messages with the `T_log()` function:

```
void T_log(T_verbosity verbosity, char const *fmt, ...);
```

A newline is automatically added to terminate the log message line.

Listing 8.17: Log Message Example

```
#include <t.h>

T_TEST_CASE(log)
{
    T_log(T_NORMAL, "a log message %i, %i, %i", 1, 2, 3);
    T_set_verbosity(T_QUIET);
    T_log(T_NORMAL, "not verbose enough");
}
```

Listing 8.18: Log Message Report

```
B:log
L:a log message 1, 2, 3
E:log:N:0:F:0
```

You can use the following functions to print formatted output:

```
int T_printf(char const *, ...);

int T_vprintf(char const *, va_list);

int T_snprintf(char *, size_t, const char *, ...);
```

In contrast to the corresponding standard C library functions, floating-point and exotic formats may not be supported. On some architectures supported by RTEMS, floating-point operations are only supported in special tasks and may be forbidden in interrupt context. The formatted output functions provided by the test framework work in every context.

### 8.1.10 Utility

You can stop a test case via the `T_stop()` function. This function does not return. You can indicate unreachable code paths with the `T_unreachable()` function. If this function is called, then the test case stops.

You can busy wait with the `T_busy()` function:

```
void T_busy(uint_fast32_t count);
```

It performs a busy loop with the specified iteration count. This function is optimized to not perform memory accesses and should have a small jitter. The loop iterations have a processor-specific duration.

You can get an iteration count for the `T_busy()` function which corresponds roughly to one clock tick interval with the `T_get_one_clock_tick_busy()` function:

```
uint_fast32_t T_get_one_clock_tick_busy(void);
```

This function requires a clock driver. It must be called from thread context with interrupts enabled. It may return a different value each time it is called.

### 8.1.11 Time Services

The test framework provides two unsigned integer types for time values. The `T_ticks` unsigned integer type is used by the `T_tick()` function which measures time using the highest frequency counter available on the platform. It should only be used to measure small time intervals. The `T_time` unsigned integer type is used by the `T_now()` function which returns the current monotonic clock value of the platform, e.g. `CLOCK_MONOTONIC`.

```
T_ticks T_tick(void);

T_time T_now(void);
```

The reference time point for these two clocks is unspecified. You can obtain the test case begin time with the `T_case_begin_time()` function.

```
T_time T_case_begin_time(void);
```

You can convert time into ticks with the `T_time_to_ticks()` function and vice versa with the `T_ticks_to_time()` function.

```
T_time T_ticks_to_time(T_ticks ticks);

T_ticks T_time_to_ticks(T_time time);
```

You can convert seconds and nanoseconds values into a combined time value with the `T_seconds_and_nanoseconds_to_time()` function. You can convert a time value into separate seconds and nanoseconds values with the `T_time_to_seconds_and_nanoseconds()` function.

```
T_time T_seconds_and_nanoseconds_to_time(uint32_t s, uint32_t ns);

void T_time_to_seconds_and_nanoseconds(T_time time, uint32_t *s, uint32_t *ns);
```

You can convert a time value into a string represention. The time unit of the string representation is seconds. The precision of the string represention may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (`T_time_string`).

```
const char *T_time_to_string_ns(T_time time, T_time_string buffer);

const char *T_time_to_string_us(T_time time, T_time_string buffer);

const char *T_time_to_string_ms(T_time time, T_time_string buffer);

const char *T_time_to_string_s(T_time time, T_time_string buffer);
```

Listing 8.19: Time String Example

```
#include <t.h>

T_TEST_CASE(time_to_string)
{
    T_time_string ts;
    T_time t;
    uint32_t s;
    uint32_t ns;

    t = T_seconds_and_nanoseconds_to_time(0, 123456789);
    T_eq_str(T_time_to_string_ns(t, ts), "0.123456789");
    T_eq_str(T_time_to_string_us(t, ts), "0.123456");
    T_eq_str(T_time_to_string_ms(t, ts), "0.123");
    T_eq_str(T_time_to_string_s(t, ts), "0");

    T_time_to_seconds_and_nanoseconds(t, &s, &ns);
    T_eq_u32(s, 0);
```

(continues on next page)

(continued from previous page)

```
    T_eq_u32(ns, 123456789);
}
```

Listing 8.20: Time String Report

```
B:time_to_string
P:0:0:UI1:test-time.c:11
P:1:0:UI1:test-time.c:12
P:2:0:UI1:test-time.c:13
P:3:0:UI1:test-time.c:14
P:4:0:UI1:test-time.c:17
P:5:0:UI1:test-time.c:18
E:time_to_string:N:6:F:0:D:0.005250
```

You can convert a tick value into a string represention. The time unit of the string representation is seconds. The precision of the string represention may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (`T_time_string`).

```
const char *T_ticks_to_string_ns(T_ticks ticks, T_time_string buffer);

const char *T_ticks_to_string_us(T_ticks ticks, T_time_string buffer);

const char *T_ticks_to_string_ms(T_ticks ticks, T_time_string buffer);

const char *T_ticks_to_string_s(T_ticks ticks, T_time_string buffer);
```

### 8.1.12 Code Runtime Measurements

You can measure the runtime of code fragments in several execution environment variants with the `T_measure_runtime()` function. This function needs a context which must be created with the `T_measure_runtime_create()` function. The context is automatically destroyed after the test case execution.

```
typedef struct {
    size_t sample_count;
} T_measure_runtime_config;

typedef struct {
    const char *name;
    int flags;
    void (*setup)(void *arg);
    void (*body)(void *arg);
    bool (*teardown)(void *arg, T_ticks *delta, uint32_t tic, uint32_t toc,
        unsigned int retry);
    void *arg;
} T_measure_runtime_request;

T_measure_runtime_context *T_measure_runtime_create(
    const T_measure_runtime_config *config);

```

(continues on next page)

(continued from previous page)

```
18 void T_measure_runtime(T_measure_runtime_context *ctx,
19     const T_measure_runtime_request *request);
```

The runtime measurement is performed for the body request handler of the measurement request (T_measure_runtime_request). The optional setup request handler is called before each invocation of the body request handler. The optional teardown request handler is called after each invocation of the body request handler. It has several parameters and a return status. If it returns true, then this measurement sample value is recorded, otherwise the measurement is retried. The delta parameter is the current measurement sample value. It can be altered by the teardown request handler. The tic and toc parameters are the system tick values before and after the request body invocation. The retry parameter is the current retry counter. The runtime of the operational setup and teardown request handlers is not measured.

You can control some aspects of the measurement through the request flags (use zero for the default):

**T_MEASURE_RUNTIME_ALLOW_CLOCK_ISR**
Allow clock interrupts during the measurement. By default, measurements during which a clock interrupt happened are discarded unless it happens two times in a row.

**T_MEASURE_RUNTIME_REPORT_SAMPLES**
Report all measurement samples.

**T_MEASURE_RUNTIME_DISABLE_FULL_CACHE**
Disable the FullCache execution environment variant.

**T_MEASURE_RUNTIME_DISABLE_HOT_CACHE**
Disable the HotCache execution environment variant.

**T_MEASURE_RUNTIME_DISABLE_DIRTY_CACHE**
Disable the DirtyCache execution environment variant.

**T_MEASURE_RUNTIME_DISABLE_MINOR_LOAD**
Disable the Load execution environment variants with a load worker count less than the processor count.

**T_MEASURE_RUNTIME_DISABLE_MAX_LOAD**
Disable the Load execution environment variant with a load worker count equal to the processor count.

The execution environment variants (M:V) are:

**FullCache**
Before the body request handler is invoked a memory area with twice the size of the outermost data cache is completely read. This fills the data cache with valid cache lines which are unrelated to the body request handler. The cache is full with valid data and loading memory used by the handler needs to evict cache lines.

You can disable this variant with the T_MEASURE_RUNTIME_DISABLE_FULL_CACHE request flag.

**HotCache**
Before the body request handler is invoked the body request handler is called without measuring the runtime. The aim is to load all data used by the body request handler to the cache.

You can disable this variant with the T_MEASURE_RUNTIME_DISABLE_HOT_CACHE request flag.

**DirtyCache**

Before the body request handler is invoked a memory area with twice the size of the outer-most data cache is completely written with new data. This should produce a data cache with dirty cache lines which are unrelated to the body request handler. In addition, the entire instruction cache is invalidated.

You can disable this variant with the T_MEASURE_RUNTIME_DISABLE_DIRTY_CACHE request flag.

**Load/<WorkerCount>**

This variant tries to get close to worst-case conditions. The cache is set up according to the DirtyCache variant. In addition, other processors try to fully load the memory system. The load is produced through writes to a memory area with twice the size of the outer-most data cache. The load variant is performed multiple times with a different set of active load worker threads. The <WorkerCount> value is the count of active workers which ranges from one to the processor count.

You can disable these variants with the T_MEASURE_RUNTIME_DISABLE_MINOR_LOAD and T_MEASURE_RUNTIME_DISABLE_MAX_LOAD request flags.

On SPARC, the body request handler is called with a register window setting so that window overflow traps will occur in the next level function call.

Each execution in an environment variant produces a sample set of body request handler runtime measurements. The minimum (M:MI), first quartile (M:Q1), median (M:Q2), third quartile (M:Q3), maximum (M:MX), median absolute deviation (M:MAD), and the sum of the sample values (M:D) is reported.

Listing 8.21: Code Runtime Measurement Example

```
#include <t.h>

static void
empty(void *arg)
{
    (void)arg;
}

T_TEST_CASE(measure_empty)
{
    static const T_measure_runtime_config config = {
        .sample_count = 1024
    };
    T_measure_runtime_context *ctx;
    T_measure_runtime_request req;

    ctx = T_measure_runtime_create(&config);
    T_assert_not_null(ctx);

    memset(&req, 0, sizeof(req));
    req.name = "Empty";
    req.body = empty;
    T_measure_runtime(ctx, &req);
}
```

Listing 8.22: Code Runtime Measurement Report

```
B:measure_empty
P:0:0:UI1:test-rtems-measure.c:18
M:B:Empty
M:V:FullCache
M:N:1024
M:MI:0.000000000
M:Q1:0.000000000
M:Q2:0.000000000
M:Q3:0.000000000
M:MX:0.000000009
M:MAD:0.000000000
M:D:0.000000485
M:E:Empty:D:0.208984183
M:B:Empty
M:V:HotCache
M:N:1024
M:MI:0.000000003
M:Q1:0.000000003
M:Q2:0.000000003
M:Q3:0.000000003
M:MX:0.000000006
M:MAD:0.000000000
M:D:0.000002626
M:E:Empty:D:0.000017046
M:B:Empty
M:V:DirtyCache
M:N:1024
M:MI:0.000000007
M:Q1:0.000000007
M:Q2:0.000000007
M:Q3:0.000000008
M:MX:0.000000559
M:MAD:0.000000000
M:D:0.000033244
M:E:Empty:D:1.887834875
M:B:Empty
M:V:Load/1
M:N:1024
M:MI:0.000000000
M:Q1:0.000000002
M:Q2:0.000000002
M:Q3:0.000000003
M:MX:0.000000288
M:MAD:0.000000000
M:D:0.000002421
M:E:Empty:D:0.001798809
[... 22 more load variants ...]
M:E:Empty:D:0.021252583
```

(continues on next page)

(continued from previous page)

```
M:B:Empty
M:V:Load/24
M:N:1024
M:MI:0.000000001
M:Q1:0.000000002
M:Q2:0.000000002
M:Q3:0.000000003
M:MX:0.000001183
M:MAD:0.000000000
M:D:0.000003406
M:E:Empty:D:0.015188063
E:measure_empty:N:1:F:0:D:14.284869
```

### 8.1.13 Interrupt Tests

In the operating system implementation you may have two kinds of critical sections. Firstly, there are low-level critical sections protected by interrupts disabled and maybe also some SMP spin lock. Secondly, there are high-level critical sections which are protected by disabled thread dispatching. The high-level critical sections may contain several low-level critical sections. Between these low-level critical sections interrupts may happen which could alter the code path taken in the high-level critical section.

The test framework provides support to write test cases for high-level critical sections though the `T_interrupt_test()` function:

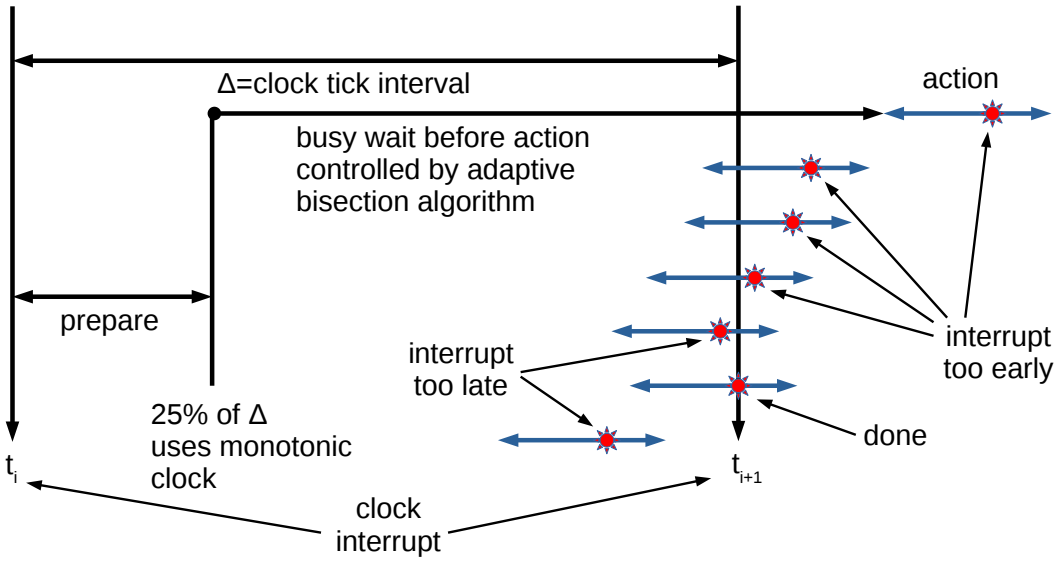
```
typedef enum {
    T_INTERRUPT_TEST_INITIAL,
    T_INTERRUPT_TEST_ACTION,
    T_INTERRUPT_TEST_BLOCKED,
    T_INTERRUPT_TEST_CONTINUE,
    T_INTERRUPT_TEST_DONE,
    T_INTERRUPT_TEST_EARLY,
    T_INTERRUPT_TEST_INTERRUPT,
    T_INTERRUPT_TEST_LATE,
    T_INTERRUPT_TEST_TIMEOUT
} T_interrupt_test_state;

typedef struct {
    void                   (*prepare)(void *arg);
    void                   (*action)(void *arg);
    T_interrupt_test_state (*interrupt)(void *arg);
    void                   (*blocked)(void *arg);
    uint32_t                 max_iteration_count;
} T_interrupt_test_config;

T_interrupt_test_state T_interrupt_test(
    const T_interrupt_test_config *config,
    void                          *arg
);
```

This function returns `T_INTERRUPT_TEST_DONE` if the test condition was satisfied within the max-

imum iteration count, otherwise it returns `T_INTERRUPT_TEST_TIMEOUT`. The interrupt test run uses the specified configuration and passes the specified argument to all configured handlers. The function shall be called from thread context with interrupts enabled.




The interrupt test uses an *adaptive bisection algorithm* to try to hit the code section under test by an interrupt. In each test iteration, it waits for a time point one quarter of the clock tick interval after a clock tick using the monotonic clock. Then it performs a busy wait using `T_busy()` with a busy count controlled by the adaptive bisection algorithm. The test maintains a sample set of upper and lower bound busy wait count values. Initially, the lower bound values are zero and the upper bound values are set to a value returned by `T_get_one_clock_tick_busy()`. The busy wait count for an iteration is set to the middle point between the arithmetic mean of the lower and upper bound sample values. After the action handler returns, the set of lower and upper bound sample values is updated based on the test state. If the test state is `T_INTERRUPT_TEST_EARLY`, then the oldest upper bound sample value is replaced by the busy wait count used to delay the action and the latest lower bound sample value is slightly decreased. Reducing the lower bound helps to avoid a zero length interval between the upper and lower bounds. If the test state is `T_INTERRUPT_TEST_LATE`, then the oldest lower bound sample value is replaced by the busy wait count used to delay the action and the latest upper bound sample value is slightly increased. In all other test states the timing values remain as is. Using the arithmetic mean of a sample set dampens the effect of each test iteration and is an heuristic to mitigate the influence of jitters in the action code execution.

The optional *prepare* handler should prepare the system so that the *action* handler can be called. It is called in a tight loop, so all the time consuming setup should be done before `T_interrupt_test()` is called. During the preparation the test state is `T_INTERRUPT_TEST_INITIAL`. The preparation handler shall not change the test state.

The *action* handler should call the function which executes the code section under test. The execution path up to the code section under test should have a low jitter. Otherwise, the adaptive bisection algorithm may not find the right spot.

The *interrupt* handler should check if the test condition is satisfied or a new iteration is necessary. This handler is called in interrupt context. It shall return `T_INTERRUPT_TEST_DONE` if the test condition is satisfied and the test run is done. It shall return `T_INTERRUPT_TEST_EARLY` if the interrupt happened too early to satisfy the test condition. It shall return `T_INTERRUPT_TEST_LATE` if the interrupt happened too late to satisfy the test condition. It shall

return `T_INTERRUPT_TEST_CONTINUE` if the test should continue with the current timing settings. Other states shall not be returned. It is critical to return the early and late states if the test condition was not satisfied, otherwise the adaptive bisection algorithm may not work. The returned state is used to try to change the test state from `T_INTERRUPT_TEST_ACTION` to the returned state.

The optional *blocked* handler is invoked if the executing thread blocks during the action processing. It should remove the blocking condition of the thread so that the next iteration can start. It can use `T_interrupt_change_state()` to change the interrupt test state.

The *max iteration count* configuration member defines the maximum iteration count of the test loop. If the maximum iteration count is reached before the test condition is satisfied, then `T_interrupt_test()` returns `T_INTERRUPT_TEST_TIMEOUT`.

The *interrupt* and *blocked* handlers may be called in arbitrary test states.

The *action*, *interrupt*, and *blocked* handlers can use `T_interrupt_test_get_state()` to get the current test state:

```
T_interrupt_test_state T_interrupt_test_get_state(void);
```

The *action*, *interrupt*, and *blocked* handlers can use `T_interrupt_test_change_state()` to try to change the test state from an expected state to a desired state:

```
T_interrupt_test_state T_interrupt_test_change_state(
    T_interrupt_test_state expected_state,
    T_interrupt_test_state desired_state
);
```

The function returns the previous state. If it **differs from the expected state**, then the requested state **change to the desired state did not take place**. In an SMP configuration, do not call this function in a tight loop. It could lock up the test run. To busy wait for a state change, use `T_interrupt_test_get_state()`.

The *action* handler can use `T_interrupt_test_busy_wait_for_interrupt()` to busy wait for the interrupt:

```
void T_interrupt_test_busy_wait_for_interrupt(void);
```

This is useful if the action code does not block to wait for the interrupt. If the action handler just returns the test code immediately prepares the next iteration and may miss an interrupt which happens too late.

### 8.1.14 Test Runner

You can call the `T_main()` function to run all registered test cases.

```
int T_main(const T_config *config);
```

The `T_main()` function returns 0 if all test cases passed, otherwise it returns 1. Concurrent execution of the `T_main()` function is undefined behaviour.

You can ask if you execute within the context of the test runner with the `T_is_runner()` function:

```
bool T_is_runner(void);
```

It returns `true` if you execute within the context of the test runner (the context which executes for example `T_main()`). Otherwise it returns `false`, for example if you execute in another task, in interrupt context, nobody executes `T_main()`, or during system initialization on another processor.

On RTEMS, you have to register the test cases with the `T_register()` function before you call `T_main()`. This makes it possible to run low level tests, for example without the operating system directly in `boot_card()` or during device driver initialization. On other platforms, the `T_register()` is a no operation.

```
void T_register(void);
```

You can run test cases also individually. Use `T_run_initialize()` to initialize the test runner. Call `T_run_all()` to run all or `T_run_by_name()` to run specific registered test cases. Call `T_case_begin()` to begin a freestanding test case and call `T_case_end()` to finish it. Finally, call `T_run_finalize()`.

```
void T_run_initialize(const T_config *config);

void T_run_all(void);

void T_run_by_name(const char *name);

void T_case_begin(const char *name, const T_fixture *fixture);

void T_case_end(void);

bool T_run_finalize(void);
```

The `T_run_finalize()` function returns `true` if all test cases passed, otherwise it returns `false`. Concurrent execution of the runner functions (including `T_main()`) is undefined behaviour. The test suite configuration must be persistent throughout the test run.

```
typedef enum {
    T_EVENT_RUN_INITIALIZE,
    T_EVENT_CASE_EARLY,
    T_EVENT_CASE_BEGIN,
    T_EVENT_CASE_END,
    T_EVENT_CASE_LATE,
    T_EVENT_RUN_FINALIZE
} T_event;

typedef void (*T_action)(T_event, const char *);

typedef void (*T_putchar)(int, void *);

typedef struct {
    const char *name;
    char *buf;
    size_t buf_size;
    T_putchar putchar;
    void *putchar_arg;
```

(continues on next page)

(continued from previous page)

```
    T_verbosity verbosity;
    T_time (*now)(void);
    size_t action_count;
    const T_action *actions;
} T_config;
```

With the test suite configuration you can specifiy the test suite name, the put character handler used the output the test report, the initial verbosity, the monotonic time provider and an optional set of test suite actions. Only the test runner calls the put character handler, other tasks or interrupt handlers write to a buffer which is emptied by the test runner on demand. You have to specifiy this buffer in the test configuration. The test suite actions are called with the test suite name for test suite run events (`T_EVENT_RUN_INITIALIZE` and `T_EVENT_RUN_FINALIZE`) and the test case name for the test case events (`T_EVENT_CASE_EARLY`, `T_EVENT_CASE_BEGIN`, `T_EVENT_CASE_END` and `T_EVENT_CASE_LATE`).

### 8.1.15 Test Verbosity

Three test verbosity levels are defined:

**T_QUIET**
Only the test suite begin, system, test case end, and test suite end lines are printed.

**T_NORMAL**
Prints everything except passed test lines.

**T_VERBOSE**
Prints everything.

The test verbosity level can be set within the scope of one test case with the `T_set_verbosity()` function:

```
T_verbosity T_set_verbosity(T_verbosity new_verbosity);
```

The function returns the previous verbosity. After the test case, the configured verbosity is automatically restored.

An example with `T_QUIET` verbosity:

```
A:xyz
S:Platform:RTEMS
[...]
E:a:N:2:F:1
E:b:N:0:F:1
E:c:N:1:F:1
E:d:N:6:F:0
Z:xyz:C:4:N:9:F:3
```

The same example with `T_NORMAL` verbosity:

```
A:xyz
S:Platform:RTEMS
[...]
B:a
```

(continues on next page)

(continued from previous page)

```
F:1:0:UI1:test-verbosity.c:6:test fails
E:a:N:2:F:1
B:b
F:*:0:UI1:test-verbosity.c:12:quiet test fails
E:b:N:0:F:1
B:c
F:0:0:UI1:test-verbosity.c:17:this is a format string
E:c:N:1:F:1
B:d
E:d:N:6:F:0
Z:xyz:C:4:N:9:F:3
```

The same example with `T_VERBOSE` verbosity:

```
A:xyz
S:Platform:RTEMS
[...]
B:a
P:0:0:UI1:test-verbosity.c:5
F:1:0:UI1:test-verbosity.c:6:test fails
E:a:N:2:F:1
B:b
F:*:0:UI1:test-verbosity.c:12:quiet test fails
E:b:N:0:F:1
B:c
F:0:0:UI1:test-verbosity.c:17:this is a format string
E:c:N:1:F:1
B:d
P:0:0:UI1:test-verbosity.c:22
P:1:0:UI1:test-verbosity.c:23
P:2:0:UI1:test-verbosity.c:24
P:3:0:UI1:test-verbosity.c:25
P:4:0:UI1:test-verbosity.c:26
P:5:0:UI1:test-verbosity.c:27
E:d:N:6:F:0
Z:xyz:C:4:N:9:F:3
```

### 8.1.16 Test Reporting

The test reporting is line based which should be easy to parse with a simple state machine. Each line consists of a set of fields separated by colon characters (`:`). The first character of the line determines the line format:

**A**

A test suite begin line. It has the format:

**A:<TestSuite>**

A description of the field follows:

**<TestSuite>**
The test suite name. Must not contain colon characters (`:`).

**S**

A test suite system line. It has the format:

**S:<Key>:<Value>**

A description of the fields follows:

**<Key>**
A key string. Must not contain colon characters (`:`).

**<Value>**
An arbitrary key value string. May contain colon characters (`:`).

**B**

A test case begin line. It has the format:

**B:<TestCase>**

A description of the field follows:

**<TestCase>**
A test case name. Must not contain colon characters (`:`).

**P**

A test pass line. It has the format:

**P:<Step>:<Processor>:<Task>:<File>:<Line>**

A description of the fields follows:

**<Step>**
Each non-quiet test has a unique test step counter value in each test case execution. The test step counter is set to zero before the test case executes. For quiet test checks, there is no associated test step and the character `*` instead of an integer is used to indicate this.

**<Processor>**
The processor index of the processor which executed at least one instruction of the corresponding test.

**<Task>**
The name of the task which executed the corresponding test if the test executed in task context. The name `ISR` indicates that the test executed in interrupt context. The name `?` indicates that the test executed in an arbitrary context with no valid executing task.

**<File>**
The name of the source file which contains the corresponding test. A source file of `*` indicates that no test source file is associated with the test, e.g. it was produced by the test framework itself.

**<Line>**
The line of the test statement in the source file which contains the corresponding test. A line number of `*` indicates that no test source file is associated with the test, e.g. it was produced by the test framework itself.

**F**

A test failure line. It has the format:

**F:<Step>:<Processor>:<Task>:<File>:<Line>:<Message>**

A description of the fields follows:

**<Step> <Processor> <Task> <File> <Line>**
See above **P** line.

**<Message>**
An arbitrary message string. May contain colon characters (:).

**L**

A log message line. It has the format:

**L:<Message>**

A description of the field follows:

**<Message>**
An arbitrary message string. May contain colon characters (:).

**E**

A test case end line. It has the format:

**E:<TestCase>:N:<Steps>:F:<Failures>:D:<Duration>**

A description of the fields follows:

**<TestCase>**
A test case name. Must not contain colon characters (:).

**<Steps>**
The final test step counter of a test case. Quiet test checks produce no test steps.

**<Failures>**
The count of failed test checks of a test case.

**<Duration>**
The test case duration in seconds.

**Z**

A test suite end line. It has the format:

**Z:<TestSuite>:C:<TestCases>:N:<OverallSteps>:F:<OverallFailures>:D:<Duration>**

A description of the fields follows:

**<TestSuite>**
The test suite name. Must not contain colon characters (:).

**<TestCases>**
The count of test cases in the test suite.

**<OverallSteps>**
The overall count of test steps in the test suite.

**<OverallFailures>**
The overall count of failed test cases in the test suite.

**<Duration>**
The test suite duration in seconds.

**Y**

Auxiliary information line. Issued after the test suite end. It has the format:

**Y:ReportHash:SHA256:<Hash>**

A description of the fields follows:

**<Hash>**
The SHA256 hash value of the test suite report from the begin to the end of the test suite.

**M**
A code runtime measurement line. It has the formats:

**M:B:<Name>**

**M:V:<Variant>**

**M:N:<SampleCount>**

**M:S:<Count>:<Value>**

**M:MI:<Minimum>**

**M:Q1:<FirstQuartile>**

**M:Q2:<Median>**

**M:Q3:<ThirdQuartile>**

**M:MX:<Maximum>**

**M:MAD:<MedianAbsoluteDeviation>**

**M:D:<SumOfSampleValues>**

**M:E:<Name>:D:<Duration>**

A description of the fields follows:

**<Name>**
A code runtime measurement name. Must not contain colon characters (:).

**<Variant>**
The execution variant which is one of **FullCache**, **HotCache**, **DirtyCache**, or **Load/<WorkerCount>**. The <WorkerCount> is the count of active workers which ranges from one to the processor count.

**<SampleCount>**
The sample count as defined by the runtime measurement configuration.

**<Count>**
The count of samples with the same value.

**<Value>**
A sample value in seconds.

**<Minimum>**
The minimum of the sample set in seconds.

**<FirstQuartile>**
The first quartile of the sample set in seconds.

**<Median>**
The median of the sample set in seconds.

**<ThirdQuartile>**
The third quartile of the sample set in seconds.

**<Maximum>**
The maximum of the sample set in seconds.

**<MedianAbsoluteDeviation>**
The median absolute deviation of the sample set in seconds.

**<SumOfSampleValues>**
The sum of all sample values of the sample set in seconds.

**<Duration>**
The runtime measurement duration in seconds. It includes time to set up the execution environment variant.

Listing 8.23: Example Test Report

```
A:xyz
S:Platform:RTEMS
S:Compiler:7.4.0 20181206 (RTEMS 5, RSB e0aec65182449a4e22b820e773087636edaf5b32,␣
↪Newlib 1d35a003f)
S:Version:5.0.0.820977c5af17c1ca2f79800d64bd87ce70a24c68
S:BSP:erc32
S:RTEMS_DEBUG:1
S:RTEMS_MULTIPROCESSING:0
S:RTEMS_POSIX_API:1
S:RTEMS_PROFILING:0
S:RTEMS_SMP:1
B:timer
P:0:0:UI1:test-rtems.c:26
P:1:0:UI1:test-rtems.c:29
P:2:0:UI1:test-rtems.c:33
P:3:0:ISR:test-rtems.c:14
P:4:0:ISR:test-rtems.c:15
P:5:0:UI1:test-rtems.c:38
P:6:0:UI1:test-rtems.c:39
P:7:0:UI1:test-rtems.c:42
E:timer:N:8:F:0:D:0.019373
B:rsc_success
P:0:0:UI1:test-rtems.c:59
F:1:0:UI1:test-rtems.c:60:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
F:*:0:UI1:test-rtems.c:62:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
P:2:0:UI1:test-rtems.c:63
F:3:0:UI1:test-rtems.c:64:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
E:rsc_success:N:4:F:3:D:0.011128
B:rsc
P:0:0:UI1:test-rtems.c:48
F:1:0:UI1:test-rtems.c:49:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
F:*:0:UI1:test-rtems.c:51:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
P:2:0:UI1:test-rtems.c:52
F:3:0:UI1:test-rtems.c:53:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
E:rsc:N:4:F:3:D:0.011083
Z:xyz:C:3:N:16:F:6:D:0.047201
Y:ReportHash:SHA256:e5857c520dd9c9b7c15d4a76d78c21ccc46619c30a869ecd11bbcd1885155e0b
```

### 8.1.17 Test Report Validation

You can add the `T_report_hash_sha256()` test suite action to the test suite configuration to generate and report the SHA256 hash value of the test suite report. The hash value covers everything reported by the test suite run from the begin to the end. This can be used to check that the report generated on the target is identical to the report received on the report consumer side. The hash value is reported after the end of test suite line (`Z`) as auxiliary information in a `Y` line. Consumers may have to reverse a `\\n` to `\\r\\n` conversion before the hash is calculated. Such a conversion could be performed by a particular put character handler provided by the test suite configuration.

### 8.1.18 Supported Platforms

The framework runs on FreeBSD, MSYS2, Linux and RTEMS.

## 8.2 Test Framework Requirements for RTEMS

The requirements on a test framework suitable for RTEMS are:

### 8.2.1 License Requirements

**TF.License.Permissive**
The test framework shall have a permissive open source license such as BSD-2-Clause.

### 8.2.2 Portability Requirements

**TF.Portability**
The test framework shall be portable.

**TF.Portability.RTEMS**
The test framework shall run on RTEMS.

**TF.Portability.POSIX**
The test framework shall be portable to POSIX compatible operating systems. This allows to run test cases of standard C/POSIX/etc. APIs on multiple platforms.

**TF.Portability.POSIX.Linux**
The test framework shall run on Linux.

**TF.Portability.POSIX.FreeBSD**
The test framework shall run on FreeBSD.

**TF.Portability.C11**
The test framework shall be written in C11.

**TF.Portability.Static**
Test framework shall not use dynamic memory for basic services.

**TF.Portability.Small**
The test framework shall be small enough to support low-end platforms (e.g. 64KiB of RAM/ROM should be sufficient to test the architecture port, e.g. no complex stuff such as file systems, etc.).

**TF.Portability.Small.LinkTimeConfiguration**
The test framework shall be configured at link-time.

**TF.Portability.Small.Modular**
The test framework shall be modular so that only necessary parts end up in the final executable.

**TF.Portability.Small.Memory**
The test framework shall not aggregate data during test case executions.

### 8.2.3 Reporting Requirements

**TF.Reporting**
Test results shall be reported.

**TF.Reporting.Verbosity**
The test report verbosity shall be configurable. This allows different test run scenarios, e.g. regression test runs, full test runs with test report verification against the planned test output.

**TF.Reporting.Verification**
It shall be possible to use regular expressions to verify test reports line by line.

**TF.Reporting.Compact**
Test output shall be compact to avoid long test runs on platforms with a slow output device, e.g. 9600 Baud UART.

**TF.Reporting.PutChar**
A simple output one character function provided by the platform shall be sufficient to report the test results.

**TF.Reporting.NonBlocking**
The ouptut functions shall be non-blocking.

**TF.Reporting.Printf**
The test framework shall provide printf()-like output functions.

**TF.Reporting.Printf.WithFP**
There shall be a printf()-like output function with floating point support.

**TF.Reporting.Printf.WithoutFP**
There shall be a printf()-like output function without floating point support on RTEMS.

**TF.Reporting.Platform**
The test platform shall be reported.

**TF.Reporting.Platform.RTEMS.Git**
The RTEMS source Git commit shall be reported.

**TF.Reporting.Platform.RTEMS.Arch**
The RTEMS architecture name shall be reported.

**TF.Reporting.Platform.RTEMS.BSP**
The RTEMS BSP name shall be reported.

**TF.Reporting.Platform.RTEMS.Tools**
The RTEMS tool chain version shall be reported.

**TF.Reporting.Platform.RTEMS.Config.Debug**
The shall be reported if RTEMS_DEBUG is defined.

**TF.Reporting.Platform.RTEMS.Config.Multiprocessing**
The shall be reported if RTEMS_MULTIPROCESSING is defined.

**TF.Reporting.Platform.RTEMS.Config.POSIX**
The shall be reported if RTEMS_POSIX_API is defined.

**TF.Reporting.Platform.RTEMS.Config.Profiling**
The shall be reported if RTEMS_PROFILING is defined.

**TF.Reporting.Platform.RTEMS.Config.SMP**
The shall be reported if RTEMS_SMP is defined.

**TF.Reporting.TestCase**
The test cases shall be reported.

**TF.Reporting.TestCase.Begin**
The test case begin shall be reported.

**TF.Reporting.TestCase.End**
The test case end shall be reported.

**TF.Reporting.TestCase.Tests**
The count of test checks of the test case shall be reported.

**TF.Reporting.TestCase.Failures**
The count of failed test checks of the test case shall be reported.

**TF.Reporting.TestCase.Timing**
Test case timing shall be reported.

**TF.Reporting.TestCase.Tracing**
Automatic tracing and reporting of thread context switches and interrupt service routines shall be optionally performed.

### 8.2.4 Environment Requirements

**TF.Environment**
The test framework shall support all environment conditions of the platform.

**TF.Environment.SystemStart**
The test framework shall run during early stages of the system start, e.g. valid stack pointer, initialized data and cleared BSS, nothing more.

**TF.Environment.BeforeDeviceDrivers**
The test framework shall run before device drivers are initialized.

**TF.Environment.InterruptContext**
The test framework shall support test case code in interrupt context.

### 8.2.5 Usability Requirements

**TF.Usability**
The test framework shall be easy to use.

**TF.Usability.TestCase**
It shall be possible to write test cases.

**TF.Usability.TestCase.Independence**
It shall be possible to write test cases in modules independent of the test runner.

**TF.Usability.TestCase.AutomaticRegistration**
Test cases shall be registered automatically, e.g. via constructors or linker sets.

**TF.Usability.TestCase.Order**
It shall be possible to sort the registered test cases (e.g. random, by name) before they are executed.

**TF.Usability.TestCase.Resources**
It shall be possible to use resources with a life time restricted to the test case.

**TF.Usability.TestCase.Resources.Memory**
It shall be possible to dynamically allocate memory which is automatically freed once the test case completed.

**TF.Usability.TestCase.Resources.File**
It shall be possible to create a file which is automatically unlinked once the test case completed.

**TF.Usability.TestCase.Resources.Directory**
It shall be possible to create a directory which is automatically removed once the test case completed.

**TF.Usability.TestCase.Resources.FileDescriptor**
It shall be possible to open a file descriptor which is automatically closed once the test case completed.

**TF.Usability.TestCase.Fixture**
It shall be possible to use a text fixture for test cases.

**TF.Usability.TestCase.Fixture.SetUp**
It shall be possible to provide a set up handler for each test case.

**TF.Usability.TestCase.Fixture.TearDown**
It shall be possible to provide a tear down handler for each test case.

**TF.Usability.TestCase.Context**
The test case context shall be verified a certain points.

**TF.Usability.TestCase.Context.VerifyAtEnd**
After a test case exection it shall be verified that the context is equal to the context at the test case begin. This helps to ensure that test cases are independent of each other.

**TF.Usability.TestCase.Context.VerifyThread**
The test framework shall provide a function to ensure that the test case code executes in normal thread context. This helps to ensure that operating system service calls return to a sane context.

**TF.Usability.TestCase.Context.Configurable**
The context verified in test case shall be configurable at link-time.

**TF.Usability.TestCase.Context.ThreadDispatchDisableLevel**
It shall be possible to verify the thread dispatch disable level.

**TF.Usability.TestCase.Context.ISRNestLevel**
It shall be possible to verify the ISR nest level.

**TF.Usability.TestCase.Context.InterruptLevel**
It shall be possible to verify the interrupt level (interrupts enabled/disabled).

**TF.Usability.TestCase.Context.Workspace**
It shall be possible to verify the workspace.

**TF.Usability.TestCase.Context.Heap**
It shall be possible to verify the heap.

**TF.Usability.TestCase.Context.OpenFileDescriptors**
It shall be possible to verify the open file descriptors.

**TF.Usability.TestCase.Context.Classic**
It shall be possible to verify Classic API objects.

**TF.Usability.TestCase.Context.Classic.Barrier**
It shall be possible to verify Classic API Barrier objects.

**TF.Usability.TestCase.Context.Classic.Extensions**
It shall be possible to verify Classic API User Extensions objects.

**TF.Usability.TestCase.Context.Classic.MessageQueues**
It shall be possible to verify Classic API Message Queue objects.

**TF.Usability.TestCase.Context.Classic.Partitions**
It shall be possible to verify Classic API Partition objects.

**TF.Usability.TestCase.Context.Classic.Periods**
It shall be possible to verify Classic API Rate Monotonic Period objects.

**TF.Usability.TestCase.Context.Classic.Regions**
It shall be possible to verify Classic API Region objects.

**TF.Usability.TestCase.Context.Classic.Semaphores**
It shall be possible to verify Classic API Semaphore objects.

**TF.Usability.TestCase.Context.Classic.Tasks**
It shall be possible to verify Classic API Task objects.

**TF.Usability.TestCase.Context.Classic.Timers**
It shall be possible to verify Classic API Timer objects.

**TF.Usability.TestCase.Context.POSIX**
It shall be possible to verify POSIX API objects.

**TF.Usability.TestCase.Context.POSIX.Keys**
It shall be possible to verify POSIX API Key objects.

**TF.Usability.TestCase.Context.POSIX.KeyValuePairs**
It shall be possible to verify POSIX API Key Value Pair objects.

**TF.Usability.TestCase.Context.POSIX.MessageQueues**
It shall be possible to verify POSIX API Message Queue objects.

**TF.Usability.TestCase.Context.POSIX.Semaphores**
It shall be possible to verify POSIX API Named Semaphores objects.

**TF.Usability.TestCase.Context.POSIX.Shms**
It shall be possible to verify POSIX API Shared Memory objects.

**TF.Usability.TestCase.Context.POSIX.Threads**
It shall be possible to verify POSIX API Thread objects.

**TF.Usability.TestCase.Context.POSIX.Timers**
It shall be possible to verify POSIX API Timer objects.

**TF.Usability.Assert**
There shall be functions to assert test objectives.

**TF.Usability.Assert.Safe**
Test assert functions shall be safe to use, e.g. assert(a == b) vs. assert(a = b) vs. assert_eq(a, b).

**TF.Usability.Assert.Continue**
There shall be assert functions which allow the test case to continue in case of an assertion failure.

**TF.Usability.Assert.Abort**
There shall be assert functions which abourt the test case in case of an assertion failure.

**TF.Usability.EasyToWrite**
It shall be easy to write test code, e.g. avoid long namespace prefix rtems_test_*.

**TF.Usability.Threads**
The test framework shall support multi-threading.

**TF.Usability.Pattern**
The test framework shall support test patterns.

**TF.Usability.Pattern.Interrupts**
The test framework shall support test cases which use interrupts, e.g. spintrcritical*.

**TF.Usability.Pattern.Parallel**
The test framework shall support test cases which want to run code in parallel on SMP machines.

**TF.Usability.Pattern.Timing**
The test framework shall support test cases which want to measure the timing of code sections under various platform conditions, e.g. dirty cache, empty cache, hot cache, with load from other processors, etc. . .

**TF.Usability.Configuration**
The test framework shall be configurable.

**TF.Usability.Configuration.Time**
The timestamp function shall be configurable, e.g. to allow test runs without a clock driver.

### 8.2.6 Performance Requirements

**TF.Performance.RTEMS.No64BitDivision**
The test framework shall not use 64-bit divisions on RTEMS.

## 8.3 Off-the-shelf Test Frameworks

There are several off-the-shelf test frameworks for C/C++. The first obstacle for test frameworks is the license requirement (`TF.License.Permissive`).

### 8.3.1 bdd-for-c

In the bdd-for-c framework the complete test suite must be contained in one file and the main function is generated. This violates `TF.Usability.TestCase.Independence`.

### 8.3.2 CBDD

The CBDD framework uses the C blocks extension from clang. This violates `TF.Portability.C11`.

### 8.3.3 Google Test

Google Test 1.8.1 was supported by RTEMS. Unfortunately, it is written in C++ and is too heavy weight for low-end platforms. Otherwise it is a nice framework. We have archived it in case someone wants to try to bring it back.

### 8.3.4 Unity

The Unity Test API does not meet our requirements. There was a discussion on the mailing list in 2013.

## 8.4 Standard Test Report Formats

### 8.4.1 JUnit XML

A common test report format is JUnit XML.

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites id="xyz" name="abc" tests="225" failures="1262" time="0.001">
  <testsuite id="def" name="ghi" tests="45" failures="17" time="0.001">
    <testcase id="jkl" name="mno" time="0.001">
      <failure message="pqr" type="stu"></failure>
      <system-out>stdout</system-out>
      <system-err>stderr</system-err>
    </testcase>
  </testsuite>
</testsuites>
```

The major problem with this format is that you have to output the failure count of all test suites and the individual test suite before the test case output. You know the failure count only after a complete test run. This runs contrary to requirement `TF.Portability.Small.Memory`. It is also a bit verbose (`TF.Reporting.Compact`).

It is easy to convert a full test report generated by *The RTEMS Test Framework* (page 182) to the JUnit XML format.

### 8.4.2 Test Anything Protocol

The Test Anything Protocol (TAP) is easy to consume and produce.

```
1..4
ok 1 - Input file opened
not ok 2 - First line of the input valid
ok 3 - Read the rest of the file
not ok 4 - Summarized correctly # TODO Not written yet
```

You have to know in advance how many test statements you want to execute in a test case. The problem with this format is that there is no standard way to provide auxiliary data such as test timing or a tracing report.

It is easy to convert a full test report generated by *The RTEMS Test Framework* (page 182) to the TAP format.

CHAPTER

**NINE**

# FORMAL VERIFICATION

## 9.1 Formal Verification Overview

Formal Verification is a technique based on writing key design artifacts using notations that have a well-defined mathematical *semantics*. This means that these descriptions can be rigorously analyzed using logic and other mathematical tools. The term *formal model* is used to refer to any such description.

Having a formal model of a software engineering artifact (requirements, specification, code) allows it to be analyzed to assess the behavior it describes. This means checks can be done that the model has desired properties, and that it lacks undesired ones. A key feature of having a formal description is that tools can be developed that parse the notation and perform much, if not most, of the analysis. An industrial-strength formalism is one that has very good tool support.

Having two formal models of the same software object at different levels of abstraction (specification and code, say) allows their comparison. In particular, a formal analysis can establish if a lower level artifact like code satisfies the properties described by a higher level, such as a specification. This relationship is commonly referred to as a *refinement*.

Often it is quite difficult to get a useful formal model of real code. Some formal modelling approaches are capable of generating machine-readable *scenarios* that describe possible correct behaviors of the system at the relevant level of abstraction. A refinement for these can be defined by using them to generate test code. This is the technique that is used in *Test Generation Methodology* (page 230) to verify parts of RTEMS. Formal models are constructed based on requirements documentation, and are used as a basis for test generation.

## 9.2 Formal Verification Approaches

This is an overview of a range of formal methods and tools that look feasible for use with RTEMS.

A key criterion for any proposed tool is the ability to deploy it in a highly automated manner. This amounts to the tool having a command-line interface that covers all the required features. One such feature is that the tool generates output that can be easily transformed into the formats useful for qualification. Tools with GUI interfaces can be very helpful while developing and deploying formal models, as long as the models/tests/proofs can be re-run automatically via the command-line.

Other important criteria concerns the support available for test generation support, and how close the connection is between the formalism and actual C code.

The final key criteria is whatever techniques are proposed should fit in with the RTEMS Project Mission Statement, in the Software Engineering manual. This requires, among other things, that any tool added to the tool-chain needs to be open-source.

A more detailed report regarding this can be found in [BH21].

Next is a general overview of formal methods and testing, and discusses a number of formalisms and tools against the criteria above.

### 9.2.1 Formal Methods Overview

Formal specification languages can be divided into the following groups:

Model-based: e.g., Z, VDM, B

> These have a language that describes a system in terms of having an abstract state and how it is modified by operations. Reasoning is typically based around the notions of pre- and post-conditions and state invariants. The usual method of reasoning is by using theorem-proving. The resulting models often have an unbounded number of possible states, and are capable of describing unbounded numbers of operation steps.

Finite State-based: e.g., finite-state machines (FSMs), SDL, Statecharts

> These are a variant of model-based specification, with the added constraint that the number of states are bounded. Desired model properties are often expressed using some form of temporal logic. The languages used to describe these are often more constrained than in more general model-based approaches. The finiteness allows reasoning by searching the model, including doing exhaustive searches, a.k.a. model-checking.

Process Algebras: e.g., CSP, CCS, pi-calculus, LOTOS

> These model systems in terms of the sequence of externally observable events that they perform. There is no explicit definition of the abstract states, but their underlying semantics is given as a state machine, where the states are deduced from the overall behavior of the system, and events denote transitions between these states. In general both the number of such states and length of observed event sequences are unbounded. While temporal logics can be used to express properties, many process algebras use their own notation to express desired properties by simpler systems.

A technique called bisimulation is used to reason about the relationships between these.

Most of the methods above start with formal specifications/models. Also needed is a way to bridge the gap to actual code. The relationship between specification and code is often referred to as a *refinement* (some prefer the term *reification*). Most model-based methods have refinement, with the concept baked in as a key part of the methodology.

Theorem Provers: e.g., CoQ, HOL4, PVS, Isabelle/HOL

Many modern theorem provers are not only useful to help reason about the formalisms mentioned above, but are often powerful enough to be used to describe formal models in their own terms and then apply their proof systems directly to those.

Model Checkers: e.g., SPIN, FDR

Model checkers are tools that do exhaustive searches over models with a finite number of states. These are most commonly used with the finite-state methods, as well as the process algebras where some bound is put on the state-space. As model-checking is basically exhaustive testing, these are often the easiest way to get test generation from formal techniques.

Formal Development frameworks: e.g. TLA+, Frama-C, KeY

There are also a number of frameworks that support a close connection between a programming language, a formalism to specify desired behavior for programs in that language, as well as tools to support the reasoning (proof, simulation, test).

### 9.2.2 Formal Methods actively considered

Given the emphasis on verifying RTEMS C code, the focus is on freely available tools that could easily connect to C. These include: Frama-C, TLA+/PlusCal, Isabelle/HOL, and Promela/SPIN. Further investigation ruled out TLA+/PlusCal because it is Java-based, and requires installing a Java Runtime Environment. Frama-C, Isabelle/HOL, and Promela/SPIN are discussed below in more detail,

#### 9.2.2.1 Frama-C

Frama-C (frama-c.com) is a platform supporting a range of tools for analysing C code, including static analysers, support for functional specifications (ANSI-C Specification Language – ACSL), and links to theorem provers. Some of its analyses require code annotations, while others can extract useful information from un-annotated code. It has a plug-in architecture, which makes it easy to extend. It is used extensively by Airbus.

Frama-C, and its plugins, are implemented in OCaml, and it is installed using the `opam` package manager. An issue here was that Frama-C has many quite large dependencies. There was support for test generation, but it was not freely available. Another issue was that Frama-C only supported C99, and not C11 (the issue is how to handle C11 Atomics in terms of their semantics).

#### 9.2.2.2 Isabelle/HOL

Isabelle/HOL is a wide-spectrum theorem-prover, implemented as an embedding of Higher-Order Logic (HOL) into the Isabelle generic proof assistant (isabelle.in.tum.de). It has a high degree of automation, including an ability to link to third-party verification tools, and a very large library of verified mathematical theorems, covering number and set theory, algebra, analysis. It is based on the idea of a small trusted code kernel that defines an encapsulated datatype representing a theorem, which can only be constructed using methods in the kernel for that datatype, but which also scales effectively regardless of how many theorems are so proven. It is implemented using `polyml`, with the IDE implemented using Scala, is open-source, and is easy to install. However, like Frama-C, it is also a very large software suite.

### 9.2.3 Formal Method actually used

A good survey of formal techniques and testing is found in a 2009 ACM survey paper [HBB+09]. Here they clearly state:

> "The most important role for formal verification in testing is in the automated generation of test cases. In this context, model checking is the formal verification technology of choice; this is due to the ability of model checkers to produce counterexamples in case a temporal property does not hold for a system model."

#### 9.2.3.1 Promela/SPIN

The current use of formal methods in RTEMS is based on using the Promela language to model key RTEMS features, in such a way that tests can be generated using the SPIN model checker (spinroot.com). Promela is quite a low-level modelling language that makes it easy to get close to code level, and is specifically targeted to modelling software. It is one of the most widely used model-checkers, both in industry and education. It uses assertions, and *Linear Temporal Logic* (*LTL*) to express properties of interest.

Given a Promela model that checks key properties successfully, tests can be generated for a property *P* by asking SPIN to check the negation of that property. There are ways to get SPIN to generate multiple/all possible counterexamples, as well as getting it to find the shortest.

## 9.3 Test Generation Methodology

The general approach to using any model-checking technology for test generation has three major steps:

### 9.3.1 Model desired behavior

Construct a model that describes the desired properties (`P1`, ..., `PN`) and use the model-checker to verify those properties.

Promela can specify properties using the `assert()` statement, to be true at the point where it gets executed, and can use *Linear Temporal Logic* (LTL) to specify more complex properties over execution sequences. SPIN will also check generic correctness properties such as deadlock and livelock freedom.

### 9.3.2 Make claims about undesired behavior

Given a fully verified model, systematically negate each specified property. Given that each property was verified as true, then these negated properties will fail model-checking, and counter-examples will be generated. These counter-examples will in fact be scenarios describing correct behavior of the system, demonstrating the truth of each property.

**Warning**

It is very important that the negations only apply to stated properties, and do not alter the possible behaviors of the model in any way. The behaviours of the model are determined by the control-flow constructs, so any boolean-valued expression statements used in these, or used in sequential code to wait for some some condition, should not be altered. What can be altered are the expressions in `assert()` statements, and any LTL properties.

With Promela, there are a number of different ways to do systematic negation. The precise approach adopted depends on the nature of the models, and more details can be found in the RTEMS Formal Models Guide Appendix in this document.

### 9.3.3 Map good behavior scenarios to tests

Define a mapping from counter-example output to test code, and use this in the process of constructing a test program.

A YAML file is used to define a mapping from SPIN output to relevant fragments of RTEMS C test code, using the Test Framework section in this document. The process is automated by a python script called `testbuilder`.

## 9.4 Formal Tools Setup

The required formal tools consist of the model checking software (Promela/SPIN), and the test generation software (spin2test/testbuilder).

### 9.4.1 Installing Tools

#### 9.4.1.1 Installing Promela/SPIN

Follow the installation instructions for Promela/Spin at https://spinroot.com/spin/Man/README.html.

There are references there to the Spin Distribution which is now on Github (https://github.com/nimble-code/Spin).

#### 9.4.1.2 Installing Test Generation Tools

The test generation tools are found in `formal/promela/src`, written in Python3, and installed using a virtual environment. To build the tools, enter `formal/promela/src` and issue the commands:

```
make env
. env/bin/activate
make py
```

The test generation tools need to be used from within this Python virtual environment. Use the `deactivate` command to exit from it.

Test generation is managed at the top level by the script `testbuilder.py` located in the top-level of `formal/promela/src`. To avoid using (long) absolute pathnames, it helps to define an suitable alias *(e.g.)*:

```
alias tbuild='python3 /..../formal/promela/src/testbuilder.py'
```

This alias is used subsequently in this documentation.

To check for a successful tool build, invoke the command without any arguments, which should result in an extended help message being displayed:

```
(env) prompt % tbuild
USAGE:
help - more details about usage and commands below
all modelname - runs clean, spin, gentests, copy, compile and run
clean modelname - remove spin, test files
archive modelname - archives spin, test files
zero  - remove all tesfiles from RTEMS
spin modelname - generate spin files
gentests modelname - generate test files
copy modelname - copy test files and configuration to RTEMS
compile - compiles RTEMS tests
run - runs RTEMS tests
```

The tool is not yet ready for use, as it needs to be configured.

### 9.4.2 Tool Configuration

Tool configuration involves setting up a new testsuite in RTEMS, and providing information to `tbuild` that tells it where to find key locations, and some command-line arguments for some of the tools. A template file `testbuilder-template.yml` is included, and contains the following entries:

```
# This should be specialised for your setup, as testbuilder.yml,
# located in the same directory as testbuilder.py
# All pathnames should be absolute

spin2test: <spin2test_directory>/spin2test.py
rtems: <path-to-main-rtems-directory>  # rtems.git, or ..../modules/rtems/
rsb: <rsb-build_directory>/rtems/6/bin/
simulator: <path-to>/sparc-rtems6-sis
testyamldir: <rtems>/spec/build/testsuites/validation/ # directory containing
↪<modelname>.yml
testcode: <rtems>/testsuites/validation/
testexedir:  <rtems>/build/.../testsuites/validation/ # directory containing ts-
↪<modelname>.exe
testsuite: model-0
simulatorargs: -leon3 -r s -m 2  # run command executes "<simulator> <simargs>
↪<testexedir>/ts-<testsuite>.exe"
spinallscenarios: -DTEST_GEN -run -E -c0 -e # trail generation "spin
↪<spinallscenarios> <model>.pml"
```

This template should be copied/renamed to `testbuilder.yml` and each entry updated as follows:

- **spin2test:**
  This should be the absolute path to `spin2test.py` in the Promela sources directory.

  `/.../formal/promela/src/spin2test.py`

- **rtems:**
  This should be the absolute path to your RTEMS source directory, with the terminating `/`. From `rtems-central` this would be:

  `/.../rtems-central/modules/rtems/`

  For a separate `rtems` installation it would be where `rtems.git` was cloned.

  We refer to this path below as `<rtems>`.

- **rsb:**
  This should be the absolute path to your RTEMS source-builder binaries directory, with the terminating `/`. From `rtems-central` this would be (assuming RTEMS 6):

  `/.../rtems-central/modules/rsb/6/bin/`

- **simulator:**
  This should be the absolute path to the RTEMS Tester (See Host Tools in the RTEMS User Manual)

  It defaults at present to the `sis` simulator

  `/.../rtems-central/modules/rsb/6/bin/sparc-rtems6-sis`

- **testsuite:**
  This is the name for the testsuite :

  Default value: `model-0`

- **testyamldir:**
  This should be the absolute path to where validation tests are *specified*:

  `<rtems>/spec/build/testsuites/validation/`

- **testcode:**
  This should be the absolute path to where validation test sources are found:

  `<rtems>/testsuites/validation/`

- **testexedir:**
  This should be the absolute path to where the model-based validation test executable will be found:

  `<rtems>/build/.../testsuites/validation/`

  This will contain `ts-<testsuite>.exe` (e.g. `ts-model-0.exe`)

- **simulatorargs:**
  These are the command line arguments for the RTEMS Tester. It defaults at present to those for the `sis` simulator.

  `-<bsp> -r s -m <cpus>`

  The first argument should be the BSP used when building RTEMS sources. BSPs `leon3`, `gr712rc` and `gr740` have been used. The argument to the `-m` flag is the number of cores. Possible values are: 1, 2 and 4 (BSP dependent)

  Default: `-leon3 -r s -m 2`

- **spinallscenarios:**
  These are command line arguments for SPIN, that ensure that all counter-examples are generated.

  Default: `-DTEST_GEN -run -E -c0 -e` (recommended)

#### 9.4.2.1 Testsuite Setup

The C test code generated by these tools is installed into the main `rtems` repository at `testsuites/validation` in the exact same way as other RTEMS test code. This means that whenever `waf` is used at the top level to build and/or run tests, that the formally generated code is automatically included. This requires adding and modifying some *Specification Items* (See Software Requirements Engineering section in this document).

To create a testsuite called `model-0` (say), do the following, in the `spec/build/testsuites/validation` directory:

- Edit `grp.yml` and add the following two lines into the `links` entry:

```
- role: build-dependency
  uid: model-0
```

- Copy `validation-0.yml` (say) to `model-0.yml`, and change the following entries as shown:

```
enabled-by: RTEMS_SMP
source:
- testsuites/validation/ts-model-0.c
target: testsuites/validation/ts-model-0.exe
```

Then, go to the `testsuites/validation` directory, and copy `ts-validation-0.c` to `ts-model-0.c`, and edit as follows:

- Change all occurrences of `Validation0` in comments to `Model0`.
- Change `rtems_test_name` to `Model0`.

### 9.4.3 Running Test Generation

The testbuilder takes a command as its first command-line argument. Some of these commands require the model-name as a second argument:

> Usage: `tbuild <command> [<modelname>]`

The commands provided are:

**`clean <model>`**
Removes generated files.

**`spin <model>`**
Runs SPIN to find all scenarios. The scenarios are found in numbered files called `<model>N.spn`.

**`gentests <model>`**
Convert SPIN scenarios to test sources. Each `<model>N.spn` produces a numbered test source file.

**`copy <model>`**
Copies the generated test files to the relevant test source directory, and updates the relevant test configuration files.

**`archive <model>`**
Copies generated spn, trail, source, and test log files to an archive sub-directory of the model directory.

**`compile`**
Rebuilds the test executable.

**`run`**
Runs tests in a simulator.

**`all <model>`**
Does clean, spin, gentests, copy, compile, and run.

**`zero`**
Removes all generated test filenames from the test configuration files, but does NOT remove the test sources from the test source directory.

In order to generate test files the following input files are required:

> `<model>.pml`, `<model>-rfn.yml`, `<model>-pre.h`, `<model>-post.h`, and `<model>-run.h`.

In addition there may be other files whose names have <model> embedded in them. These are included in what is transfered to the test source directory by the copy command.

The simplest way to check test generation is setup properly is to visit one of the models, found under `formal/promela/models` and execute the following command:

```
tbuild all mymodel
```

This should end by generating a file `model-0-test.log`. The output is identical to that generated by the regular RTEMS tests, using the *Software Test Framework* described elsewhere in this document.

Output for the Event Manager model, highly redacted:

```
SIS - SPARC/RISCV instruction simulator 2.29,  copyright Jiri Gaisler 2020
Bug-reports to jiri@gaisler.se

GR740/LEON4 emulation enabled, 4 cpus online, delta 50 clocks

Loaded ts-model-0.exe, entry 0x00000000

*** BEGIN OF TEST Model0 ***
*** TEST VERSION: 6.0.0.03337dab21e961585d323a9974c8eea6106c803d
*** TEST STATE: EXPECTED_PASS
*** TEST BUILD: RTEMS_SMP
*** TEST TOOLS: 10.3.1 20210409 (RTEMS 6, RSB␣
 ↪889cf95db0122bd1a6b21598569620c40ff2069d, Newlib eb03ac1)
A:Model0
S:Platform:RTEMS
...
B:RtemsModelSystemEventsMgr8
...
L:@@@ 3 CALL event_send 1 2 10 sendrc
L:Calling Send(167837697,10)
L:Returned 0x0 from Send
...
E:RtemsModelEventsMgr0:N:21:F:0:D:0.005648
Z:Model0:C:18:N:430:F:0:D:0.130464
Y:ReportHash:SHA256:5EeLdWsRd25IE-ZsS6pduLDsrD_qzB59dMU-Mg2-BDA=

*** END OF TEST Model0 ***

cpu 0 in error mode (tt = 0x80)
  6927700  0000d580:  91d02000   ta  0x0
```

## 9.5 Modelling with Promela

Promela is a large language with many features, but only a subset is used here for test generation. This is a short overview of that subset. The definitive documentation can be found at https://spinroot.com/spin/Man/promela.html.

### 9.5.1 Promela Execution

Promela is a *modelling* language, not a programming language. It is designed to describe the kind of runtime behaviors that make reasoning about low-level concurrency so difficult: namely shared mutable state and effectively non-deterministic interleaving of concurrent threads. This means that there are control constructs that specify non-deterministic outcomes, and an execution model that allows the specification of when threads should block.

The execution model is based on the following concepts:

**Interleaving Concurrency**
A running Promela system consists of one or more concurrent processes. Each process is described by a segment of code that defines a sequence of atomic steps. The scheduler looks at all the available next-steps and makes a **non-deterministic choice** of which one will run. The scheduler is invoked after every atomic step.

**Executability**
At any point in time, a Promela process is either able to perform a step, and is considered executable, or is unable to do so, and is considered blocked. Whether a statement is executable or blocked may depend on the global state of the model. The scheduler will only select from among the executable processes.

The Promela language is based loosely on C, and the SPIN model-checking tool converts a Promela model into a C program that has the specific model hard-coded and optimized for whatever analysis has been invoked. It also supports the use of the C pre-processor.

#### 9.5.1.1 Simulation vs. Verification

SPIN can run a model in several distinct modes:

**Simulation**
SPIN simply makes random choices for the scheduler to produce a possible execution sequence (a.k.a. scenario) allowed by the model. A readable transcript is written to `stdout` as the simulation runs.

The simplest SPIN invocation does simulation by default:

```
spin model.pml
```

**Verification**
SPIN does an analysis of the whole model by exploring all the possible choices that the scheduler can make. This will continue until either all possible choices have been covered, or some form of error is uncovered. If verification ends successfully, then this is simply reported as ok. If an error occurs, verification stops, and the sequence of steps that led to that failure are output to a so-called trail file.

The simplest way to run a verification is to give the `-run` option:

```
spin -run model.pml
```

**Replaying**

A trail file is an uninformative list of number-triples, but can be replayed in simulation mode to produce human-readable output.

```
spin -t model.pml
```

### 9.5.2 Promela Datatypes

Promela supports a subset of C scalar types (`short`, `int`), but also adds some of its own (`bit`, `bool`, `byte`, `unsigned`). It has support for one-dimensional arrays, and its own variation of the C struct concept (confusingly called a `typedef`!). It has a single enumeration type called `mtype`. There are no pointers in Promela, which means that modelling pointer usage requires the use of arrays with their indices acting as proxies for pointers.

### 9.5.3 Promela Declarations

Variables and one-dimensional arrays can be declared in Promela in much the same way as they are done in C:

```
int x, y[3] ;
```

All global variables and arrays are initialized to zero.

The identifier `unsigned` is the name of a type, rather than a modifier. It is used to declare an unsigned number variable with a given bit-width:

```
unsigned mask : 4 ;
```

Structure-like datatypes in Promela are defined using the `typedef` keyword that associates a name with what is basically a C `struct`:

```
typedef CBuffer {
  short count;
  byte buffer[8]
}

CBuffers cbuf[6];
```

Note that we can have arrays of `typedefs` that themselves contain arrays. This is the only way to get multi-dimensional arrays in Promela.

There is only one enumeration type, which can be defined incrementally. Consider the following sequence of four declarations that defines the values in `mtype` and declares two variables of that type:

```
mtype = { up, down } ;
mtype dir1;
mtype = { left, right} ;
mtype dir2;
```

This gives the same outcome with the following two declarations:

```
mtype = { left, right, up, down } ;
mtype dir1, dir2;
```

#### 9.5.3.1 Special Identifiers

The are a number of variable identifiers that have a special meaning in Promela. These all start with an underscore. We use the following:

**Process Id**
`_pid` holds the process id of the currently active process

**Process Count**
`_nr_pr` gives the number of currently active processes.

### 9.5.4 Promela Atomic Statements

**Assignment**
`x = e` where `x` is a variable and `e` is an expression.

Expression `e` must have no side-effects. An assignment is always executable. Its effect is to update the value of `x` with the current value of `e`.

**Condition Statement**
`e` where `e` is an expression

Expression `e`, used standalone as a statement, is executable if its value in the current state is non-zero. If its current value is zero, then it is blocked. It behaves like a NO-OP when executed.

**Skip**
`skip`, a keyword

`skip` is always executable, and behaves like a NO-OP when executed.

**Assertion**
`assert(e)` where `e` is an expression

An assertion is always executable. When executed, it evaluates its expression. If the value is non-zero, then it behaves like a NO-OP. If the value is zero, then it generates an assertion error and aborts further simulation/verification of the model.

**Printing**
`printf(string,args)` where `string` is a format-string and `args` are values and expressions.

A `printf` statement is completely ignored in verification mode. In simulation mode, it is always executable, and generates output to `stdout` in much the same way as in C. This is is used in a structured way to assist with test generation.

**Goto**
`goto lbl` where `lbl` is a statement label.

Promela supports labels for statements, in the same manner as C. The `goto` statement is always executable. When executed, flow of control goes to the statement labelled by `lbl:`.

**Break**
`break`, a keyword

Can only occur within a loop (`do ... od`, see below). It is always executable, and when executed performs a `goto` to the statement just after the end of the innermost enclosing loop.

### 9.5.5 Promela Composite Statements

**Sequencing**

`{ <stmt> ; <stmt> ; ... ; <stmt> }` where each `<stmt>` can be any kind of statement, atomic or composite. The sub-statements execute in sequence in the usual way.

**Selection**

```
if
:: <stmt>
:: <stmt>
...
:: <stmt>
fi
```

A selection construct is blocked if all the `<stmt>` are blocked. It is executable if at least one `<stmt>` is executable. The scheduler will make a non-deterministic choice from all of those `<stmt>` that are executable. The construct terminates when/if the chosen `<stmt>` does.

Typically, a selection statement will be a sequence of the form `g ; s1 ; ... ; sN` where `g` is an expression acting as a guard, and the rest of the statements are intended to run if `g` is non-zero. The symbol `->` plays the same syntactic role as `;`, so this allows for a more intuitive look and feel; `g -> s1 ; ... ; sN`.

If the last `<stmt>` has the form `else -> ...`, then the `else` is executable only when all the other selection statements are blocked.

**Repetition**

```
do
:: <stmt>
:: <stmt>
...
:: <stmt>
od
```

The executability rules here are the same as for Selection above. The key difference is that when/if a chosen `<stmt>` terminates, then the whole construct is re-executed, making it basically an infinite loop. The only way to exit this loop is for an active `<stmt>` to execute a `break` or `goto` statement.

A `break` statement only makes sense inside a Repetition, is always executable, and its effect is to jump to the next statement after the next `od` keyword in text order.

The selection and repetition syntax and semantics are based on Edsger Djikstra's Guarded Command Language [Dij75] .

**Atomic Composite**

`atomic{stmt}` where `stmt` is usually a (sequential) composite.

Wrapping the `atomic` keyword around a statement makes its entire execution proceed without any interference from the scheduler. Once it is executable, if the scheduler chooses it to run, then it runs to completion.

There is one very important exception: if it should block internally because some sub-statement is blocked, then the atomicity gets broken, and the scheduler is free to find some other executable process to run. When/if the sub-statement eventually becomes executable, once it gets chosen to run by the scheduler then it continues to run atomically.

**Processes**

`proctype name (args) { sequence }` where `args` is a list of zero or more typed parameter declarations, and `sequence` is a list of local declarations and statements.

This defines a process type called `name` which takes parameters defined by `args` and has the behavior defined by `sequence`. When invoked, it runs as a distinct concurrent process. Processes can be invoked explicitly by an existing process using the `run` statement, or can be setup to start automatically.

**Run**

`run name (exprs)` where `exprs` is a list of expressions that match the arguments defined the `proctype` declaration for `name`.

This is executable as long as the maximum process limit has not been reached, and immediately starts the process running. It is an atomic statement.

**Inlining**

`inline name (names) { sequence }` where `names` is a list of zero or more identifiers, and `sequence` is a list of declarations and statements.

Inlining does textual substitution, and does not represent some kind of function call. An invocation `name(texts)` gets replaced by `{ sequence }` where each occurrence of an identifier in `names` is replaced by the corresponding text in `texts`. Such an invocation can only appear in a context where a Promela statement can appear.

### 9.5.6 Promela Top-Level

At the top-level, a Promela model is a list of declarations, much like a C program. The Promela equivalent of `main` is a process called `init` that has no arguments. There must be at least one Promela process setup to be running at the start. This can be `init`, or one or more `proctypes` declared as `active`.

## 9.6 Promela to C Refinement

Promela models are more abstract than concrete C code. A rigorous link, known as a *refinement*, needs to be established from Promela to C. This is composed of two parts: manual annotations in the Promela model to make its behavior easy to identify and parse; and a refinement defined as a YAML file that maps from annotations to corresponding C code. A single refinement YAML file is associated with each Promela model.

### 9.6.1 Model Annotations

Promela `printf` statements are used in the models to output information that is used by `spin2test` to generate test code. This information is used to lookup keys in a YAML file that defines the mapping to C code. It uses a simple format that makes it easy to parse and process, and is also designed to be easily understood by a human reader. This is important because any V&V process will require this information to be carefully assessed.

#### 9.6.1.1 Annotation Syntax

Format, where $N \geq 0$:

```
@@@ <pid> <KEYWORD> <parameter1> ... <parameterN>
```

The leading `@@@` is a marker that makes it easy to filter out this information from other SPIN output.

Parameters take the following forms:

`<pid>` Promela Process Id of `proctype` generating annotation

`<word>` chunk of text containing no white space

`<name>` Promela variable/structure/constant identifier

`<type>` Promela type identifier

`<tid>` OS Task/Thread/Process Id

_ unused argument (within containers)

Each `<KEYWORD>` is associated with specific forms of parameters:

```
1  LOG <word1> ... <wordN>
2  NAME <name>
3  INIT
4  DEF <name> <value>
5  DECL <type> <name> [<value>]
6  DCLARRAY <type> <name> <value>
7  TASK <name>
8  SIGNAL <name> <value>
9  WAIT   <name> <value>
10 STATE tid <name>
11 SCALAR (<name>|_) [<index>] <value>
12 PTR <name> <value>
13 STRUCT <name>
14 SEQ <name>
15 END <name>
16 CALL <name> <value1> ... <valueN>
```

### 9.6.2 Annotation Lookup

The way that code is generated depends on the keyword in the annotation. In particular, the keyword determines how, or if, the YAML refinement file is looked up.

Direct Output - no lookup (`LOG`, `DEF`)

Keyword Refinement - lookup the `<KEYWORD>` (`NAME`, `INIT`, `SIGNAL`, `WAIT`)

Name Refinement - lookup first parameter (considered as text) (`TASK`, `DECL`, `DCLARRAY`, `PTR`, `CALL`, `SCALAR`)

The same name may appear in different contexts, and the name can be extended with a suffix of the form `_XXX` to lookup less frequent uses:

`_DCL` - A variable declaration

`_PTR` - The pointer value itself

`_FSCALAR` - A scalar that is a struct field

`_FPTR` - A pointer that is a struct field

Generally, the keyword, or name is used to lookup `mymodel-rfn.yml` to get a string result. This string typically has substitution placeholders, as used by the Python `format()` method for strings. The other parameters in the annotation are then textually substituted in, to produce a segment of test code.

### 9.6.3 Specifying Refinement

Using the terminology of the *The RTEMS Test Framework* (page 182) each Promela model is converted into a set of Test Cases, one for each complete scenario produced by test generation. There are a number of template files, tailored for each model, that are used to assemble the test code sources, along with code segments for each Promela process, based on the annotations output for any given scenario.

The refinement mapping from annotations to code is defined in a YAML file that describes a Python dictionary that maps a name to some C code, with placeholders that are used to allow for substituting in actual test values.

The YAML file has entries of the following form:

```
entity: |
  C code line1{0}
  ...
  C code lineM{2}
```

The entity is a reference to an annotation concept, which can refer to key declarations, values, variables, types, API calls or model events. There can be zero or more arguments in the annotations, and any occurrence of braces enclosing a number in the C code means that the corresponding annotation argument string is substituted in (using the python string object `format()` method).

The general pattern is working through all the annotations in order. The code obtained by looking up the YAML file is collated into different code-segments, one for each Promela process id (`<pid>`).

In addition to the explicit annotations generated by the Promela models, there are two implicit annotations produced by the python refinement code. These occur when the `<pid>` part

of a given explicit annotation is different to the one belonging to the immediately preceding annotation. This indicates a point in a test generation scenario where one task is suspended and another resumes. This generates internal annotations with keywords SUSPEND and WAKEUP which should have entries in the refinement file to provide code to ensure that the corresponding RTEMS tasks in the test behave accordingly.

The annotations can also be output as comments into the generated test-code, so it is easy to check that parameters are correct, and the generated code is correct.

If a lookup fails, a C comment line is output stating the lookup failed. The translation continues in any case.

#### 9.6.3.1 Lookup Example

Consider the following annotation, from the Events Manager model:

```
@@@ 1 CALL event_send 1 2 10 sendrc
```

This uses Name refinement so a lookup with event_send as the key and gets back the following text:

```
1 T_log( T_NORMAL, "Calling Send(%d,%d)", mapid( ctx, {1}), {2} );
2 {3} = ( *ctx->send )( mapid( ctx, {1} ), {2} );
3 T_log( T_NORMAL, "Returned 0x%x from Send", {3} );
```

Arguments 1, 2, 10, and sendrc are then substituted to get the code:

```
1 T_log( T_NORMAL, "Calling Send(%d,%d)", mapid( ctx, 2), 10 );
2 sendrc = ( *ctx->send )( mapid( ctx, 2 ), 10 );
3 T_log( T_NORMAL, "Returned 0x%x from Send", sendrc );
```

Given a Promela process id of 1, this code is put into a code segment for the corresponding RTEMS task.

### 9.6.4 Annotation Refinement Guide

This guide describes how each annotation is processed by the test generation software.

#### 9.6.4.1 LOG

**LOG <word1> ... <wordN> (Direct Output)**
Generate a call to T_log() with a message formed from the <word..> parameters. This message will appear in the test output for certain verbosity settings.

#### 9.6.4.2 NAME

**NAME <name> (Keyword Refinement)**
Looks up NAME (currently ignoring <name>) and returns the resulting text as is as part of the code. This code should define the name of the testcase, if needed.

#### 9.6.4.3 INIT

**INIT (Keyword Refinement)**
Lookup INIT and expect to obtain test initialisation code.

#### 9.6.4.4 TASK

**TASK <name> (Name Refinement)**
Lookup <name> and return corresponding C code.

#### 9.6.4.5 SIGNAL

**SIGNAL <value> (Keyword Refinement)**
Lookup SIGNAL and return code with <value> substituted in.

#### 9.6.4.6 WAIT

**WAIT <value> (Keyword Refinement)**
Lookup WAIT and return code with <value> substituted in.

#### 9.6.4.7 DEF

**DEF <name> <value> (Direct Output)**
Output #define <name> <value>.

#### 9.6.4.8 DECL

**DECL <type> <name> [<value>] (Name Refinement)**
Lookup <name>_DCL and substitute <name> in. If <value> is present, append =<value>. Add a final semicolon. If the <pid> value is zero, prepend static.

#### 9.6.4.9 DCLARRAY

**DCLARRAY <type> <name> <value> (Name Refinement)**
Lookup <name>_DCL and substitute <name> and <value> in. If the <pid> value is zero, prepend static.

#### 9.6.4.10 CALL

**CALL <name> <value0> .. <valueN> (Name refinement, N < 6)**
Lookup <name> and substitute all <value..> in.

#### 9.6.4.11 STATE

**STATE <tid> <name> (Name Refinement)**
Lookup <name> and substitute in <tid>.

#### 9.6.4.12 STRUCT

**STRUCT <name>**
Declares the output of the contents of variable <name> that is itself a structure. The <name> is noted, as is the fact that a structured value is being processes. Should not occur if already be processing a structure or a sequence.

#### 9.6.4.13 SEQ

**SEQ <name>**
Declares the output of the contents of array variable <name>. The <name> is noted, as is the fact that an array is being processed. Values are accumulated in a string now initialsed to empty. Should not occur if already processing a structure or a sequence.

#### 9.6.4.14 PTR

**PTR <name> <value> (Name Refinement)**
If not inside a STRUCT, lookup <name>_PTR. Two lines of code should be returned. If the <value> is zero, use the first line, otherwise use the second with <value> substituted in. This is required to handle NULL pointers.

If inside a STRUCT, lookup <name>_FPTR. Two lines of code should be returned. If the <value> is zero, use the first line, otherwise use the second with <value> substituted in. This is required to handle NULL pointers.

#### 9.6.4.15 SCALAR

There are quite a few variations here.

**SCALAR _ <value>**
Should only be used inside a SEQ. A space and <value> is appended to the string being accumulated by this SEQ.

**SCALAR <name> <value> (Name Refinement)**
If not inside a STRUCT, lookup <name>, and substitute <value> into the resulting code.

If inside a STRUCT, lookup <name>_FSCALAR and substitute the saved STRUCT name and <value> into the resulting code.

This should not be used inside a SEQ.

**SCALAR <name> <index> <value> (Name Refinement)**
If not inside a STRUCT, lookup <name>, and substitute <index> and <value> into the resulting code.

If inside a STRUCT, lookup <name>_FSCALAR and substitute the saved STRUCT name and <value> into the resulting code.

This should not be used inside a SEQ.

#### 9.6.4.16 END

**END <name>**
If inside a STRUCT, terminates processing a structured variable.

If inside a SEQ, lookup <name>_SEQ. For each line of code obtained, substitute in the saved sequence string. Terminates processing a sequence/array variable.

This should not be encountered outside of a STRUCT or SEQ.

#### 9.6.4.17 SUSPEND and WAKEUP

A change of Promela process id from oldid to newid has been found. Increment a counter that tracks the number of such changes.

SUSPEND (Keyword Refinement)

Lookup SUSPEND and substitute in the counter value, oldid and newid.

WAKEUP (Keyword Refinement)

Lookup WAKEUP and substitute in the counter value, newid and oldid.

### 9.6.5 Annotation Ordering

While most annotations occur in an order determined by any given test scenario, there are some annotations that need to be issued first. These are, in order: `NAME`, `DEF`, `DECL` and `DCLARRAY`, and finally, `INIT`.

### 9.6.6 Test Code Assembly

The code snippets produced by refining annotations are not enough. We also need boilerplate code to setup, coordinate and teardown the tests, as well as providing useful C support functions.

For a model called `mymodel` the following files are required:

- `mymodel.pml` - the Promela model
- `mymodel-rfn.yml` - the model refinement to C test code
- `tc-mymodel.c` - the testcase setup C file
- `tr-mymodel.h` - the test-runner header file
- `tr-mymodel.c` - the test-runner setup C file

The following files are templates used to assemble a single test-runner C file for each scenario generated by the Promela model:

- `mymodel-pre.h` - preamble material at the start
- `mymodel-run.h` - test runner material
- `mymodel-post.h` - postamble material at the end.

The process is entirely automated:

```
tbuild all mymodel
```

The steps of the process are as follows:

#### 9.6.6.1 Scenario Generation

When SPIN is invoked to find all scenarios, it will produce a number (N) of counterexample files with a `.trail` extension. These files hold numeric data that refer to SPIN internals.

```
mymodel.pml1.trail
...
mymodel.pmlN.trail
```

SPIN is then used to take each trail file and produce a human-readable text file, using the same format as the SPIN simulation output. SPIN numbers files from 1 up, whereas the RTEMS convention is to number things, including filenames, from zero. SPIN is used to produce readable output in text files with a `.spn` extension, with 1 subtracted from the trail file number. This results in the following files:

```
mymodel-0.spn
...
mymodel-{N-1}.spn
```

#### 9.6.6.2 Test Code Generation

Each SPIN output file `mymodel-I.spn` is converted to a C test runner file `tr-mymodel-I.c` by concatenating the following components:

**`mymodel-pre.h`**
This is a fairly standard first part of an RTEMS test C file. It is used unchanged.

**refined test segments**
The annotations in `mymodel-I.spn` are converted, in order, into test code snippets using the refinement file `mymodel-rfn.yml`. Snippets are gathered into distinct code segments based on the Promela process number reported in each annotation. Each code segment is used to construct a C function called `TestSegmentP()`, where `P` is the relevant process number.

**`mymodel-post.h`**
This is static code that declares the top-level RTEMS Tasks used in the test. These are where the code segments above get invoked.

**`mymodel-run.h`**
This defines top-level C functions that implement a given test runner. The top-level function has a name like `RtemsMyModel_Run` This is not valid C as it needs to produce a name parameterized by the relevant scenario number. It contains Python string format substitution placeholders that allow the relevant number to be added to end of the function name. So the above run function actually appears in this file as `RtemsMyModel_Run{0}`, so `I` will be substituted in for `{0}` to result in the name `RtemsMyModel_RunI`. In particular, it invokes `TestSegment0()` which contains code generated from Promela process 0, which is the main model function. This declares test variables, and initializes them.

These will generate test-runner test files as follows:

```
tr-mymodel-0.c
...
tr-mymodel-{N-1}.c
```

In addition, the test case file `tc-mymodel.c` needs to have entries added manually of the form below, for `I` in the range 0 to N-1.:

```
T_TEST_CASE( RtemsMyModelI )
{
  RtemsMyModel_RunI(
    ...
  );
}
```

These define the individual test cases in the model, each corresponding to precisely one SPIN scenario.

#### 9.6.6.3 Test Code Deployment

All files starting with `tc-` or `tr-` are copied to the relevant testsuite directory. At present, this is `testsuites/validation` at the top level in the `rtems` repository. All the names of the above files with a `.c` extension are added into a YAML file that defines the Promela generated-test sources. At present, this is `spec/build/testsuites/validation/model-0.yml` at the top-level in the `rtems` repository. They appear in the YAML file under the `source` key:

```
source:
- testsuites/validation/tc-mymodel.c
- testsuites/validation/tr-mymodel-0.c
...
- testsuites/validation/tr-mymodel-{N-1}.c
- testsuites/validation/ts-model-0.c
```

#### 9.6.6.4 Performing Tests

At this point build RTEMS as normal. e.g., with `waf`, and the tests will get built. The executable will be found in the designated build directory, *(e.g.):*

> `rtems/build/sparc/gr740/testsuites/validation/ts-model-0.exe`
>
> This can be run using the RTEMS Tester (RTEMS User Manual, Host Tools, RTEMS Tester and Run).
>
> Both building the code and running on the tester is also automated (see *Formal Tools Setup* (page 231)).

### 9.6.7 Traceability

Traceability between requirements, specifications, designs, code, and tests, is a key part of any qualification/certification effort. The test generation methodology developed here supports this in two ways, when refining an annotation:

1. If the annotation is for a declaration of some sort, the annotation itself is added as a comment to the output code, just before the refined declaration.

```
// @@@ 0 NAME Chain_AutoGen
// @@@ 0 DEF MAX_SIZE 8
#define MAX_SIZE 8
// @@@ 0 DCLARRAY Node memory MAX_SIZE
static item memory[MAX_SIZE];
// @@@ 0 DECL unsigned nptr NULL
static item * nptr = NULL;
// @@@ 0 DECL Control chain
static rtems_chain_control chain;
```

2. If the annotation is for a test of some sort, a call to `T_log()` is generated with the annotation as its text, just before the test code.

```
T_log(T_NORMAL,"@@@ 0 INIT");
rtems_chain_initialize_empty( &chain );
T_log(T_NORMAL,"@@@ 0 SEQ chain");
T_log(T_NORMAL,"@@@ 0 END chain");
show_chain( &chain, ctx->buffer );
T_eq_str( ctx->buffer, " 0" );
```

In addition to traceability, these also help when debugging models, refinement files, and the resulting test code.

CHAPTER

**TEN**

# BSP BUILD SYSTEM

The purpose of the build system is to produce and install artefacts from the RTEMS sources such as static libraries, start files, linker command files, configuration header files, header files, test programs, package description files, and third-party build system support files for a specific BSP in a user controlled configuration.

## 10.1 Goals

The system should meet the following goals:

- The install location of artefacts should be the same as in previous build systems
- Easy build configuration of BSPs through configuration options
- Enable the BSP build configuration to be placed in a version control system (e.g. no local paths)
- Fast builds (also on Windows)
- Easy to maintain, e.g. add/remove a BSP, add/change/remove configuration options, add/remove a library, add/remove an object to/from a library, add/remove tests
- Reusable build specifications (e.g. generate documentation for BSP options for the user manual)
- Validation of built artefacts (e.g. ensure that the objects are built as specified using the DWARF debug information)
- Support building of BSPs external to the project
- Customization of the build (e.g. build only a subset of the RTEMS functions)
- Support alternative compilers such as clang instead of GCC
- Ability to unit test the build system
- Version control system friendly change sets (e.g. most changes are line based in text files)

Configurable things which depend on the host computer environment such as paths to tools are intended to be passed as command line options to the `waf` command line tool. Configurable things which define what is built and how the artefacts are configured are intended to be placed in configuration files that can be configuration controlled. The `waf` build script file called `wscript` should know nothing about the layout of the sources. What is built and how it is built should be completely defined by the user configuration and the build specification items.

## 10.2 Overview

For an overview of the build system, see the *BSP Build System* chapter of the RTEMS User Manual.

## 10.3 Commands

This section explains how the *Build Item Type* (page 26) items determine what the following `waf` commands do.

### 10.3.1 BSP List

In the `./waf bsplist` command, the BSP list is generated from the *Build BSP Item Type* (page 28) items.

### 10.3.2 BSP Defaults

In the `./waf bspdefaults` command, the BSP defaults are generated from the *Build BSP Item Type* (page 28) and *Build Option Item Type* (page 34) items. Build specification items contribute to the command through the `do_defaults()` method in the `wscript`.

### 10.3.3 Configure

In the `./waf configure` command, the build specification items contribute to the command through the `prepare_configure()` and `do_configure()` methods in the `wscript`.

### 10.3.4 Build, Clean, and Install

In the `./waf`, `./waf clean`, and `./waf install` commands, the build specification items contribute to the commands through the `prepare_build()` and `do_build()` methods in the `wscript`.

## 10.4 UID Naming Conventions

Use the following patterns for *UID names* (page 17):

**abi**
Use the name `abi` for the GCC-specific ABI flags item of a BSP family. Each BSP family should have exactly one *Build Option Item Type* (page 34) item which defines the GCC-specific ABI flags for all base BSPs of the family. For an architecture named *arch* and a BSP family named *family*, the file path is `spec/build/bsps/arch/family/abi.yml`.

**abiclang**
Use the name `abiclang` for the clang-specific ABI flags item of a BSP family. Each BSP family may have at most one *Build Option Item Type* (page 34) item which defines the clang-specific ABI flags for all base BSPs of the family. For an architecture named *arch* and a BSP family named *family*, the file path is `spec/build/bsps/arch/family/abiclang.yml`.

**bsp***
Use the prefix `bsp` for base BSPs.

**cfg***
Use the prefix `cfg` for `config.h` header options.

**grp***
Use the prefix `grp` for groups.

**lib***
Use the prefix `lib` for libraries.

**linkcmds***
Use the prefix `linkcmds` for linker command files.

**obj***
Use the prefix `obj` for objects. Use

- `objmpci` for objects which are enabled by `RTEMS_MULTIPROCESSING`,
- `objnet` for objects which are enabled by `RTEMS_NETWORKING`,
- `objnetnosmp` for objects which are enabled by `RTEMS_NETWORKING` and not `RTEMS_SMP`, and
- `objsmp` for objects which are enabled by `RTEMS_SMP`.

**opt***
Use the prefix `opt` for options. Use

- `optclock*` for options which have something to do with the clock driver,
- `optconsole*` for options which have something to do with the console driver,
- `optirq*` for options which have something to do with interrupt processing,
- `optmem*` for options which have something to do with the memory configuration, map, settings, etc., and
- `optosc*` for options which have something to do with oscillators.

**start**
Use the name `start` for BSP start file items. Each architecture or BSP family should have a *Build Start File Item Type* (page 37) item which builds the start file of a BSP. For an architecture

named *arch* and a BSP family named *family*, the file path is `spec/build/bsps/arch/start.yml` or `spec/build/bsps/arch/family/start.yml`. It is preferable to have a shared start file for the architecture instead of a start file for each BSP family.

**tst***
Use the prefix `tst` for test states.

## 10.5 Build Specification Items

Specification items of refinements of the *Build Item Type* (page 26) are used by the `wscript` to determine what it should do. The `wscript` does not provide default values. All values are defined by specification items. The entry point to what is built are the *Build BSP Item Type* (page 28) items and the top-level *Build Group Item Type* (page 31) item. The user configuration defines which BSPs are built. The top-level group defaults to `/grp` and can be changed by the `--rtems-top-level` command line option given to the `waf configure` command.

The top-level group is a trade-off between the specification complexity and a strict dependency definition. Dependencies are normally explicit though the item links. However, using only explicit dependencies would make the specification quite complex, see Fig. 10.1. The top-level group and explicit *Build BSP Item Type* (page 28) items reduce the specification complexity since they use a priori knowledge of global build dependencies, see Fig. 10.2 for an example. This approach makes the build system easier, but less general.

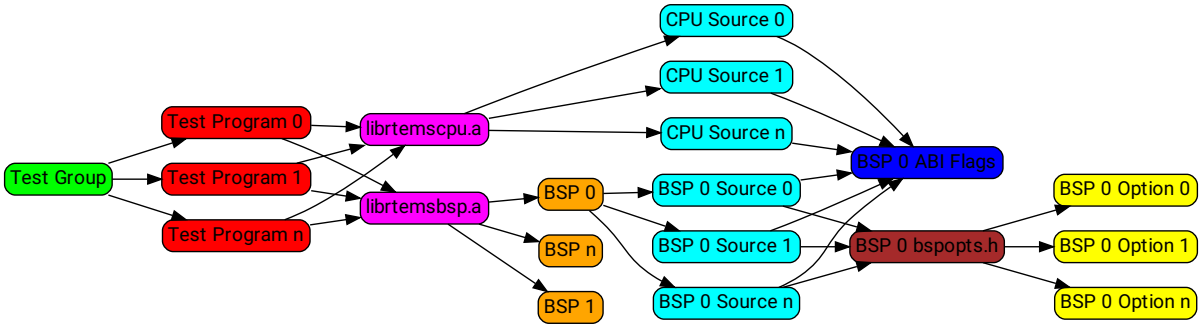



Fig. 10.1: Example with Explicit Item Links

This example shows how build item dependencies are specified explicitly by item links. In this example, a user wants to build a group of tests. Each test program has a dependency on the standard RTEMS libraries. The first issue is that the `librtemsbsp.a` needs dependencies to all base BSP variants (more than 100). The dependencies are the values of the `links` attribute in the library item files. External BSPs would have to modify the library item files. This is quite undesirable. The second issue is that the source files of the `librtemscpu.a` need a dependency to the ABI compiler flags specified by each BSP. The third issue is that each BSP has to define its own `bspopts.h` configuration header item.

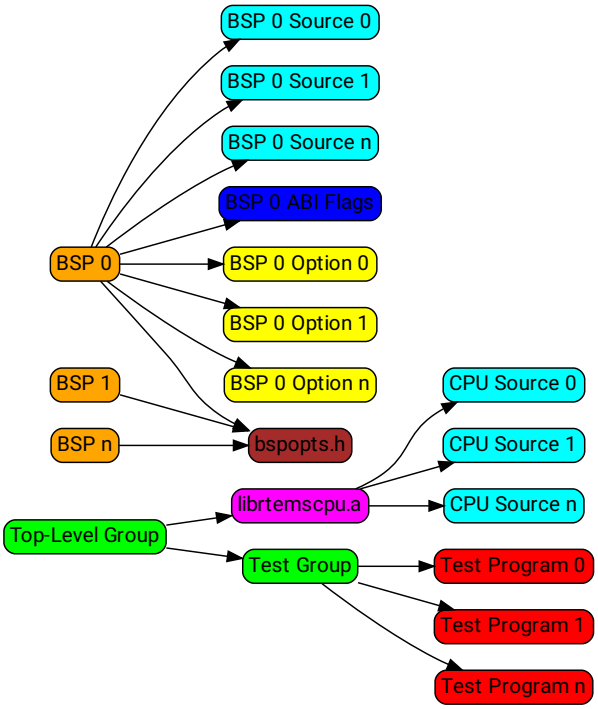



Fig. 10.2: Example with Implicit Ordering Rules

This example shows how build item dependencies are specified by dedicated BSP items, a top-level group, and ordered item links. The BSP is configured after the top-level group item and built before the top-level group item (defined by `wscript` source code). The library group is configured and built before the test group as specified by the item link order in the top-level group. The BSP options are processed before the results are written to the configuration header `bspopts.h` as defined by the BSP item link order.

## 10.6 How-To

This section presents how to do common maintenance tasks in the build system.

### 10.6.1 Find the Right Item

You find all build specification items in the RTEMS sources under the `spec/build` directory. You can use the `grep` command line tool to search in the build specification items.

### 10.6.2 Create a BSP Architecture

Let *arch* be the name of the architecture. You can add the new BSP architecture with:

```
$ mkdir spec/build/bsps/arch
```

For a new architecture try to use a shared start file which can be used by all BSPs of the architecture. Add a *Build Start File Item Type* (page 37) item for it:

```
$ vi spec/build/bsps/arch/start.yml
```

### 10.6.3 Create a BSP Family

Let *family* be the BSP family name and *arch* the name of the architecture. You can add the new BSP family with:

```
$ mkdir spec/build/bsps/arch/family
```

Add a *Build Option Item Type* (page 34) item for the ABI flags of the BSP family:

```
$ vi spec/build/bsps/arch/family/abi.yml
```

Define the ABI flags for each base BSP of the family. The `${ABI_FLAGS}` are used for the `${ASFLAGS}`, `${CFLAGS}`, `${CXXFLAGS}`, and `${LDFLAGS}` build environment variables. Please have a look at the following example which shows the GCC-specific ABI flags item of the `sparc/leon3` BSP family:

```
SPDX-License-Identifier: CC-BY-SA-4.0 OR BSD-2-Clause
actions:
- get-string: null
- split: null
- env-append: null
build-type: option
copyrights:
- Copyright (C) 2020 embedded brains GmbH & Co. KG
default:
- -mcpu=leon3
default-by-variant:
- value:
  - -mcpu=leon3
  - -mfix-ut700
  variants:
  - sparc/ut700
```

(continues on next page)

(continued from previous page)

```
- value:
  - -mcpu=leon
  - -mfix-ut699
  variants:
  - sparc/ut699
- value:
  - -mcpu=leon3
  - -mfix-gr712rc
  variants:
  - sparc/gr712rc
description: |
  ABI flags
enabled-by: gcc
links: []
name: ABI_FLAGS
type: build
```

If the architecture has no shared start file, then add a *Build Start File Item Type* (page 37) item for the new BSP family:

```
$ vi spec/build/bsps/arch/family/start.yml
```

### 10.6.4 Add a Base BSP to a BSP Family

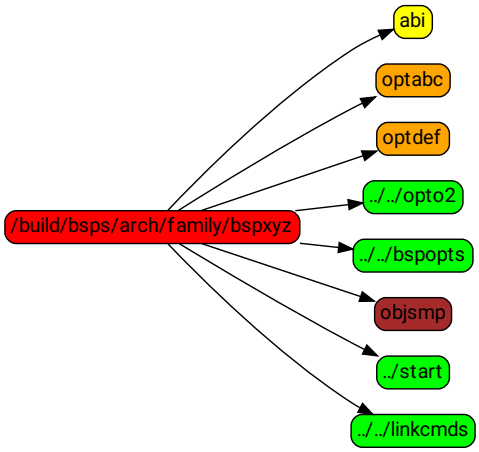



Fig. 10.3: This example shows a BSP family named *family* in the architecture *arch* which consists of only one base BSP named *xyz*. The BSP sources and installation information is contained in the `spec:/build/bsps/arch/family/bspxyz` BSP item. The items linked by the BSP item are shown using relative UIDs.

Let *family* be the BSP family name, *arch* the name of the architecture, and *new* the name of the new base BSP. You can add the new base BSP with:

```
$ vi spec/build/bsps/arch/family/bspnew.yml
```

Define the attributes of your new base BSP according to *Build BSP Item Type* (page 28).

In case the BSP family has no group, then create a group if it is likely that the BSP family will contain more than one base BSP (see *Extend a BSP Family with a Group* (page 260)).

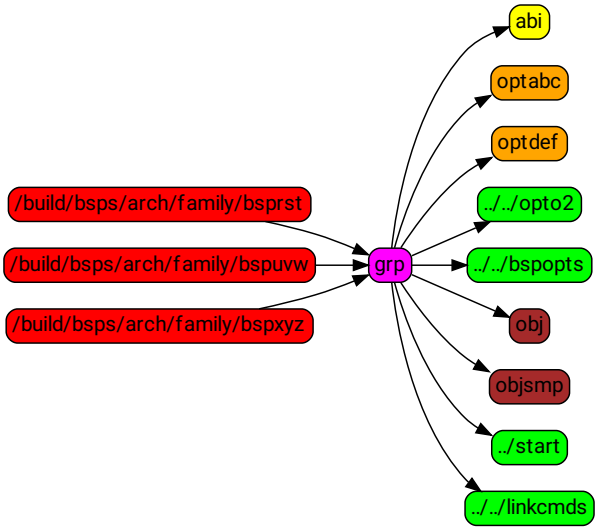



Fig. 10.4: This example shows a BSP family named *family* in the architecture *arch* which consists of three base BSPs named *rst*, *uvw*, and *xyz*. The BSP sources and installation information is contained in the *obj* objects item. The group *grp* defines the main BSP constituents. The base BSP items `spec:/build/bsps/arch/family/bsprst`, `spec:/build/bsps/arch/family/bspuvw`, and `spec:/build/bsps/arch/family/bspxyz` just define the name of the base BSP and set a link to the group item. The base BSP and BSP family names can be used for example in the `default-by-variant` attribute of *Build Option Item Type* (page 34) items. The items linked by the BSP items are shown using relative UIDs.

If the BSP family has a group, then link the new base BSP to the group with:

```
$ vi spec/build/bsps/arch/familiy/grp.yml
```

Add a link using a relative UID to the new base BSP item:

```
links:
- role: build-dependency
  uid: bspnew
```

### 10.6.5 Add a BSP Option

Let *family* be the BSP family name, *arch* the name of the architecture, and *new* the name of the new BSP option. You can add the new BSP option with:

```
$ vi spec/build/bsps/arch/family/optnew.yml
```

Define the attributes of your new BSP option according to *Build Option Item Type* (page 34). Link the option item to the corresponding group or BSP item using a relative UID:

```
links:
- role: build-dependency
  uid: optnew
```

### 10.6.6 Extend a BSP Family with a Group

Let *family* be the BSP family name and *arch* the name of the architecture. If you want to have more than one base BSP in a BSP family, then you have to use a group item (see *Build Group Item Type* (page 31)). Add the group item named *grp* to the family with:

```
$ vi spec/build/bsps/arch/family/grp.yml
```

Define the attributes of your new group according to *Build Group Item Type* (page 31) and move the links of the existing base BSP item to the group item. Add a link to *obj*.

Add an objects item named *obj* to the family with:

```
$ vi spec/build/bsps/arch/family/obj.yml
```

Define the attributes of your new objects item according to *Build Objects Item Type* (page 33) and move the `cflags`, `cppflags`, `includes`, `install` and `source` attribute values of the existing base BSP item to the objects item.

### 10.6.7 Add a Test Program

Let *collection* be the name of a test program collection and *new* the name of the new test program. You can add the new test program with:

```
$ vi spec/build/testsuites/collection/new.yml
```

Define the attributes of your new test program according to *Build Test Program Item Type* (page 38).

Edit corresponding group item of the test program collection:

```
$ vi spec/build/testsuites/collection/grp.yml
```

Add a link to the new test program item using a relative UID:

```
links:
- role: build-dependency
  uid: new
```

### 10.6.8 Add a Library

Let *new* be the name of the new library. You can add the new library with:

```
$ vi spec/build/cpukit/libnew.yml
```

Define the attributes of your new library according to *Build Library Item Type* (page 32).

Edit corresponding group item:

```
$ vi spec/build/cpukit/grp.yml
```

Add a link to the new library item using a relative UID:

```
links:
- role: build-dependency
  uid: libnew
```

### 10.6.9 Add an Object

Build objects logically separate relatively independent segments of functionality (for example a device driver, an architecture-dependent feature, etc.). Let *new* be the name of the new object. You can add the new object with:

```
$ vi spec/build/cpukit/objnew.yml
```

Define the attributes of your new object according to *Build Objects Item Type* (page 33).

Edit corresponding group item:

```
$ vi spec/build/cpukit/grp.yml
```

Add a link to the new objects item using a relative UID:

```
links:
- role: build-dependency
  uid: objnew
```

CHAPTER

**ELEVEN**

# SOFTWARE RELEASE MANAGEMENT

## 11.1 Release Process

The release process creates an RTEMS release. The process has a number of stages that happen before a release can be made, during the creation of the release and after the release has been made.

### 11.1.1 Releases

RTEMS is released as a collection of ready to use source code and built documentation. Releases are publicly available on the RTEMS servers under https://ftp.rtems.org/pub/rtems/releases.

Releases are grouped under the major version number as a collection of directories consisting of the version number. This is termed a release series. A release may also contain release candidates and snapshots.

All releases must have a three digit version number and this can be optionally followed by a dash character (`-`) and an identifier, e.g. `5.1.0-acme-1`.

The RTEMS Project reserves releases with only the three digit version number, e.g. `5.1.0`. This identifies an RTEMS Project release.

#### 11.1.1.1 Release Layout

- All released source archives are XZ compressed tar files.
- Top level contains:

**`README.txt`:**
A set of brief release links and instructions in text format generated from the `README` markdown file.

**`index.txt`:**
A set of brief release links and instructions as an HTML web page generated from the `README` markdown file.

**`contrib`:**
Contributed sources. For example the release scripts used to create the release.

**`docs`:**
Compressed documentation in HTML, Single page HTML and PDF formats. The directory contains compressed files for each document and a single archive of all the documentation. An SHA512 checksum file is also provided to allow verification of the files. The HTML documentation is available in the `docs/html` directory the `docs` directory contains the PDF documentation. Links are provided in release cover page.

**`rtems-<VERSION>-release-notes`:**
RTEMS Release notes as an HTML web site. This is a capture of the Gitlab milestone issues and merge requests in the release.

**`rtems-<VERSION>-release-notes.pdf`:**
RTEMS Release notes as a PDF document. This is a capture of the Gitlab milestone issues and merge request.

**`rtems-<VERSION>-release-notes.jzon.xz`:**
RTEMS Release JSON data captured from Gitlab for the release milestone abnd used to create the release notes.

**rtems-docs-<VERSION>.tar.xz:**
RTEMS Release Documentation source code.

**sha512sum.txt:**
SHA512 checksum of all files in this directory.

**sources:**
All source code referenced by the release.

#### 11.1.1.2 Release Version Numbering

The release numbering scheme changed with RTEMS 5. The project moved to two release numbers from the traditional three numbers. The major number was not being used and there was no easy clear process we could use to decide when to increment it. The major number role was deprecated and the numbers moved one to the left.

The RTEMS Project reserves release versions with `major.minor.0` version numbers and an empty release label. If the sources deployed to end users or systems contain changes to a release you are required to add a unique identifier to the release label.

Version string must be unique for every released version of RTEMS. The release label provides a way for deployed RTEMS sources to have a unique version string.

##### Release Number

A release number has the following fields separated by the dot (`.`) character:

**RTEMS_MAJOR**
The major version number. This number increments with each release. The value is updated after a release branch has been created.

**RTEMS_MINOR**
The minor version number is the branch release number and it increments with each release made on that release branch. The minor version number shall be `0` on all branches in the repository. The value is set using the release generated `VERSION` file.

`RTEMS_REVISION`

> The revision field is not used by the RTEMS Project and all releases it makes shall have a value of `0`. This field can used by users deploying modified releases with a suitable release label.

The main branch tracks the version `N.0.0` with `N` being the next major release number.

Examples:

- `5.0.0` is the version number of the development main for the 5 series
- `5.1.0` is the first release of the 5 series
- `5.2.0` is the first bugfix release of the 5 series
- `5.3.0` is the second bugfix release of the 5 series
- `6.0.0` is the version number of the development main for the 6 series

### Release Label

The release label is a string that can be used to provide context specific information about a release. The default value for the release label shall be `not-released`.

The users and vendors releasing RTEMS can use the release label for their own purposes. It can contain unique labels and specific versions identifiers.

The release can set the release label by:

1. A `VERSION` file that sets the release label.
2. No `VERSION` file and the sources resides in a valid version controlled repository. The release label shall be a version control system identifer that identifies a unique commit and the state of the sources under the control of the repository.
3. If there is no `VERSION` file and no valid version contolled repository found the release label shall be the default value.

A release with no release label is resevered for the RTEMS Project. This helps the project identify the origin of the release sources and how to help users with support questions.

Production builds of RTEMS from the RTEMS Projects's version controlled repository can use the version controlled identifier as a release label.

Examples the RTEMS RTOS version string:

- `6.1.0` is the version number of the first RTEMS 6 release made by the RTEMS project.
- `6.0.0.b45cf44489` is a build of RTEMS without a `VERSION` file and with the sources in a version controlled repository. The identifer is the git commit hash.
- `6.0.0.b45cf44489-modified` is the same build of source in the previous example with a locally modified file.
- `6.3.0.rc1` is the first release candidate from the second bug fix release of RTEMS 6.
- `6.1.0.acme-corp` is the vendor release from the fictional Acme Corporation based on the RTEMS 6.1.0 release.

### Version String

1. The version string is the release number and release label separated by a dash (`-`) character.
2. The RTEMS RTOS kernel version string is the release number and release label separated by a dot (`.`) character. The RTEMS version string is the only place a `.` is used to separate the version number from the release label.

#### 11.1.1.3 Release Scripts

1. The release scripts are held in the RTEMS Release repository.
2. The release scripts are not branched and the only branch is `main`. The script are maintained to make a release back to the 4.11 series.
3. The scripts are written for FreeBSD and can run on FreeBSD 10 through FreeBSD 14. No other host operating system is supported for the releases. Updates for other operating systems are welcome if the changes do not affect the operation on FreeBSD.

4. A Python `virutalenv` environment is required to runs the tools needed to make a release. The top level `README.md` file provides the specific list of packages you are required to install.
5. The release notes are generated from Issue and Merge Request data in the RTEMS Project's Gitlab instance. A read only API key is needed to create the release notes. The `README.md` provides the details about the Gitlab key and required configuration file format.
6. Building a standard release requires you provide the release major number, the release's minor number and optionally a release label:

```
./rtems-release 6 1
```

   To create a release release candidate:

```
./rtems-release 6 1-rc1
```

   To create a snapshot:

```
./rtems-release 6 0-m2410
```

7. A 3rd option of a release URL can be provided to create a test or deployable release. The URL is a base path the RSB uses to download the release source files from:

```
./rtems-release \
    -u https://ftp.rtems.org/pub/rtems/people/chrisj/releases \
    6 0.0-m2410-test
```

#### 11.1.1.4 Release Snapshots

1. Release snapshots are only created for the current development version of RTEMS. For example RTEMS 5 snapshot path is `5/5.0.0/5.0.0-m2003`.
2. Release snapshots are based on the development sources and may be unstable or not suitable for use in production.
3. A release snapshot is created each month and is named as `<major>/<version>/<version>-<YYMM>` where `YY` is the last two digits of the current year and `MM` is the month as a two digit number.
4. In the lead up to a release more than one snapshot can be created by appending `-<count>` to the snapshot version string where `<count>` is incremented starting from `1`. The first snapshot without a count is considered number `0`.
5. Release snapshots maybe removed from the RTEMS servers at the discretion of the RTEMS project

### 11.1.2 Release Repositories

The following are the repositories that a release effects. Any repository action is to be performed in the following repositories:

- `rtems.git`
- `rtems-deployment.git`
- `rtems-docs.git`

- `rtems-examples.git`
- `rtems-libbsd.git`
- `rtems-lwip.git`
- `rtems-net-legacy.git`
- `rtems-net-services.git`
- `rtems-release.git`
- `rtems-source-builder.git`
- `rtems-tools.git`
- `rtems_waf.git`

### 11.1.3 Pre-Release Procedure

1. All issues and merge requests for the release milestone must be resolved, closed, or moved to a later milestone. Issues can exist that are specific to the branch to be resolved before the first release is made.
2. Create release snapshots and post suitable build and test results.

### 11.1.4 Release Branching

A release has a release branch in each of the release repositories. A release is created from a release branch. The release branch label is the RTEMS major version number.

#### 11.1.4.1 LibBSD Release Branch

The `rtems-libbsd.git` is an exception as it has two active release branches. The repository has a release branch based on the `main` like all the release repositories and it can have a FreeBSD version specific release branch that is used in the release.

LibBSD runs two branches during it's development cycle. The `main` branch tracks the FreeBSD `current` branch. This means LibBSD tracks FreeBSD's development. LibBSD also tracks a FreeBSD branch for the RTEMS release. For example RTEMS 5 tracks FreeBSD 12 as it's release base. This provides functional stability to the RTEMS 5 release by allowing a control process to track bug fixes in FreeBSD 12.

#### 11.1.4.2 Pre-Branch Procedure

1. Create a milestone for the next version of RTEMS and for the next minor version (i.e., `.2`) after the release. To create a new milestone open an issue in https://gitlab.rtems.org/administration/gitlab If no start date is provided it will be set to the end date of the previous release in the same major series.
2. Create an Epic for the release branch named `RTEMS <Major> Release` where `<Major>` is the Major number of the release. Create two children Epics for the first two releases named `RTEMS <Major>.<Minor>` where `<Minor>` will be `.1` and `.2`.
3. All issues assigned to the release's first milestone must be resolved. Issues can exist that are specific to the release branch. Those issues must be assigned to the child Epic that matches the milestone.

4. All merge requests must be resolved. Any merge requests that remain open against the `main` branch must be set to draft status and have the milestone updated to the next major version before branching to ensure they do not accidentally land on the wrong version.
5. The following BSP must build using the RSB:
   - `arm/beagleboneblack`
6. Run the RSB command `sb-rtems-pkg` command to make sure the RSB kernel, libbsd and tools configurations reference the `main` when the branch is made.

   The RSB Git build references a specific commit so it is important the relevant configurations are valid. RSB release builds reference the source tar file in the release's `sources` directory.

#### 11.1.4.3 Branch Procedure

1. Branch labels are the major number as branch releases increment the minor number. A branch is only created when the first major release is to be made.
2. The main project repositories in Gitlab are protected so branches need to be made by a Gitlab administrator. To branch the main repositories create an issue in https://gitlab.rtems.org/administration/gitlab and provide the following list of repositories that need to be branched for the release and the commit hash in each repository to branch:
   - https://gitlab.rtems.org/rtems/docs/rtems-docs/-/branches
   - https://gitlab.rtems.org/rtems/tools/rtems-source-builder/-/branches
   - https://gitlab.rtems.org/rtems/tools/rtems-tools/-/branches
   - https://gitlab.rtems.org/rtems/rtos/rtems/-/branches
   - https://gitlab.rtems.org/rtems/pkg/rtems-libbsd/-/branches
   - https://gitlab.rtems.org/rtems/pkg/rtems-net-legacy/-/branches
   - https://gitlab.rtems.org/rtems/pkg/rtems-lwip/-/branches
   - https://gitlab.rtems.org/rtems/pkg/rtems-net-services/-/branches
   - https://gitlab.rtems.org/rtems/tools/rtems_waf/-/branches
   - https://gitlab.rtems.org/rtems/tools/rtems-deployment/-/branches
   - https://gitlab.rtems.org/rtems/rtos/rtems-examples/-/branches
   - https://gitlab.rtems.org/rtems/pkg/rtems-littlevgl/-/branches
3. Check and make sure the RSB kernel, libbsd and tools reference the branch commit.

#### 11.1.4.4 Post-Branch Procedure

1. All issues on a child epic must be resolved before that minor release is created. Resolutions may include closing as `resolution::wontfix`, closing due to an accepted merge request, or advancing the Milestone to the next release and linking the issue to the next release's child epic.
2. Create the next RC release candidate with the source as close the branch point as possible.
3. Create a ticket to clean the RSB for the release. The RSB's `main` branch carries a number of older configurations and new release configurations. These can be confusing to a new

user and add no value to a released RSB. For example leaving RTEMS 7 tool building configurations in the RTEMS 6 release.

4. Check out the release branch of `rtems-central.git`. Change all Git submodules to reference commits of the corresponding release branch. Run `./spec2modules.py`. Inspect all Git submodules for changes. If there are locally modified files, then there are two options. Firstly, integrate the changes in the release branches. Afterwards update the Git submodule commit. Secondly, change the specification so that a particular change is not made. Make sure that there are no changes after this procedure.

#### Post-Branch Version Number Updates

After the release repositories have been branched the `main` branches of some repositories have to have the major version number updated. The following is a list of the needed changes.

1. RTEMS requires the following files be changed:
   - `Doxyfile`: Update `PROJECT_NUMBER`.
   - `rtems-bsps`: Update `rtems_version`.
   - `wscript`: Update `version["__RTEMS_MAJOR__"]`.
2. RTEMS Documentation the following files be changed:
   - `wscript`: Update `rtems_major_version`.
3. RSB requires the following files be changed:
   - `source-builder/sb/version.py`: Update `_version`.
4. RTEMS Tools requires the following files be changed:
   - `config/rtems-version.ini`: Update `revision`.
   - `tester/rtems/version.cfg`: Update `rtems_version`.
5. `rtems-libbsd` requires the following files and branches be changed:
   - `README.md`: Update `Branches` section.
   - `wscript`: Update `rtems_version`.
   - Create a new branch for tracking the FreeBSD stable version, for example `6-freebsd-12`.
6. `rtems-examples` requires the following files be changed:
   - `wscript`: Update `rtems_version`.

### 11.1.5 Release Procedure

The release procedure can be performed on any FreeBSD machine and uploaded to the RTEMS FTP server. You will need ssh access to the RTEMS server `dispatch.rtems.org` and suitable permissions to write into the FTP release path on the RTEMS server.

1. The release process starts by branching the repositories. The [Branch Procedure] details how to branch the main repositories.
2. To create the RTEMS release run the release script:

```
./rtems-release <VERSION> <REVISION>
```

Example:

```
./rtems-release 6 1
```

3. Copy the release to the RTEMS FTP server:

```
ssh <user>@dispatch.rtems.org mkdir -p /data/ftp/pub/rtems/releases/<VERSION>
scp -r <VERSION>.<REVISION> <user>@dispatch.rtems.org:/data/ftp/pub/rtems/
↪releases/<VERSION>/.
```

Example:

```
ssh chrisj@dispatch.rtems.org mkdir -p /data/ftp/pub/rtems/releases/5
scp -r 5.1.0 chrisj@dispatch.rtems.org:/data/ftp/pub/rtems/releases/5/.
```

4. Verify the release has been uploaded by checking the link:

   `https://ftp.rtems.org/pub/rtems/releases/<VERSION>/<VERSION>`

5. Tag the release repositories by creating an issue in https://gitlab.rtems.org/administration/gitlab and provide the tag, the same list of repositories used to create the release branch for the release and the commit hash in each repository to tag. See the [Branch Procedure] for the list of repositories to tag.

6. Create the next release Milestone and child Epic attached to the release branch's epic. These are for the release that will follow the next release on the release branch.

### 11.1.6 Post-Release Procedure

The following procedures are performed after a release has been created.

1. Update the release to the RTEMS servers:

```
rsync --rsh=ssh -arv 6.1 chrisj@dispatch.rtems.org:/data/ftp/pub/rtems/
↪releases/6/.
```

2. Test a build of the `beagleboneblack` BSP.

### 11.1.7 VERSION File Format

1. The `VERSION` is generated when making releases by the release procedure and is contained in the relased source tar file. It shall not be placed under version control.
2. The file is in the INI format.
3. The `[DEFAULT]` section is ignored.
4. Sections not listed here are ignored.
5. The file is required to have a `[version]` section.
6. The `[version]` section is required to have a `revision` option. The revision option is a version string as defined by [Version String]. The revision label separator is a dash (-).

7. The `[version]` section can optionally contain a `release_path` option. The release path is a URL the RSB supports to the released `sources` directory. The RSB uses this field to fetch all sources used in a build.
8. An optional section `[hashes]` can be used to hold the checksums for files downloaded by the RSB. The source tar files created by the release procedure for some packages downloaded by the RSB have different checksums to the values held in the RSB repository. A checksum hash in the `VERSION` file overrides the checksum in the RSB configuration files.

Examples:

- Version only configuration:

```
[version]
revision = 6.1
```

- RSB configuration:

```
[version]
revision = 6.1
release_path = https://ftp.rtems.org/pub/rtems/releases/6/6.1/sources

[hashes]
rtems-tools-6.1.tar.xz = sha512 837d9ec058e14f26fe69a702729a7
rtems-6.1.tar.xz = sha512 b37079591a35d0601a73b32912f8773bc40
rtems-libbsd-6.1.tar.xz = sha512 768546b80cd8c8ca20fb1b695b56
```

## 11.2 Release Maintenance

The release maintenance process manages release branches that RTEMS uses to create releases. Development happens on the `main` branch and any changes for releases are managed using release branches.

These procedures are designed to work within GitLab's workflows and user interface while providing the project with control, visibility and reporting.

The milestone is used to create the release notes. The procedures are designed to preserve issues and merge requests on milestones so the release notes are an accurate account of the changes made.

### 11.2.1 Release Branch Maintenance

1. The management of release branch epics, issues and merge requests is the responsibility of all users with Developer or higher GitLab roles. Normal GitLab account holders cannot set milestones or labels, they cannot assign reviewers or promote issues to epics so issues need to be triaged before they can be worked on and resolved. Please help by triaging new issues and merge requests across all projects and repos we have when users create the issues.
2. Release branches shall only be created from a repository's `main` branch.
3. The release branch is the RTEMS version and is a number without leading zeros.
4. A tag of the form `base/<version>` where `<version>` is the version being branched shall be made to record the base commit of a release branch.
5. A release branch shall not be branched.
6. Releases are made from a release branch, and the commit on the release branch the release is made from shall be tagged with the full version string of the release.
7. APIs and features related to APIs shall not be changed on a release branch.
8. Non-overlapping additions can be made to release branches if APIs and related features are not changed. For example a network driver is added to a network stack. Community review by approvers shall determine what is suitable.
9. Development should occur on a repository's `main` branch where possible and any fix backported to a release branch using an issue attached to an epic.
10. The `main` branch shall have only one milestone, the next version's first release. For example if the next version is `7` the `main` milestone will be `7.1`.
11. A release branch shall have two milestones, the next release and the release that follows. Issues or merge requests for a release branch are assigned to the next release milestone by default and optionally moved to the following release milestone if not resolved for the next release within the release window. When a release is made a new milestone is added.
12. There is no dot zero (`.0`) release. That is reserved for the next version's development on `main` or the version of snapshots or git built versions taken from a release branch's repository.
13. The RTEMS Project only maintains and publishes releases from the previous two (2) release branches as part of its open processes. Releases from older release branches can be made under service agreements and with the support of the community.

### 11.2.2 Release Epics and Issues

Epics and issues are used to help manage the approval process for commits to release branches.

1. Every release branch shall have an associated Epic named `RTEMS <Major> Release` where `<Major>` is the branch name. This Epic shall have children Epics named `RTEMS <Major>.<Minor>` for the next two releases on the release branch.

2. Every Issue with a Milestone set to a release branch shall be linked to the Epic named `RTEMS <Major>.<Minor>` where the Major.Minor matches the Milestone.

### 11.2.3 Release Merge Requests

1. Every merge request to a release branch shall have a Milestone that matches the next release version on that branch, and shall reference or close an Issue with the same Milestone. The commit messages within the Merge Request must refer to the Issue.

2. A merge request target branch shall be the milestone's major version number. If the milestone's major number is the next RTEMS Major release the target shall be `main`.

### 11.2.4 How to Handle Backports

1. Issue triaging shall determine if an issue should be considered for backporting. Issues that are opened against one branch and requested to backport to a release branch must be cloned, which can be done with the GitLab Quick Action `/clone --with_notes` in a comment on the Issue. It is preferred to clone the notes so that the discussion/comment history leading to the backport request is preserved on the backport Issue. The milestone on the cloned issue shall be set to the requested backport release branch's next version.

2. Issues that are opened against a release branch milestone must be linked to that milestone's epic, which should be named `RTEMS <Milestone>`. Linking is accomplished by using the GitLab Quick Action `/epic rtems&<NNN>` where `<NNN>` is replaced with the Epic number for the relevant milestone's epic.

3. Issues that are determined out of scope for backporting shall be labelled `resolution::wontfix` and closed.

4. Merge Requests that target a release branch must reference or close an open Issue with a Milestone that matches the next release on that branch. It is the responsibility of submitters to ensure they close the correct Issue, and it is the responsibility of approvers to ensure that merge requests to a release branch close an open Issue with a matching milestone. When cherry-picking changes, commit messages will need to be modified locally to add the correct Issue number to the commit.

## 11.3 Software Change Report Generation

TBD - What goes here?

## 11.4 Version Description Document (VDD) Generation

TBD - discuss how generated. Preferably Dannie's project

CHAPTER

**TWELVE**

# USER'S MANUALS

TBD - write and link to useful documentation, potential URLs:

Reference the RTEMS Classic API Guide

- https://docs.rtems.org/docs/main/c-user.pdf

Reference any other existing user documentation

- https://docs.rtems.org/doxygen/main/
- https://gitlab.rtems.org/
- http://www.rtems.com/
- https://docs.rtems.org/main/

## 12.1 Documentation Style Guidelines

TBD - write me

CHAPTER

**THIRTEEN**

# LICENSING REQUIREMENTS

All artifacts shall adhere to RTEMS Project licensing requirements. Currently, the preferred licenses are:

- "Two Clause BSD" (BSD-2-Clause) for source code, and
- CC-BY-SA-4.0 license for documentation

Historically, RTEMS has been licensed under the GPL v2 with linking exception (https://www.rtems.org/license). It is preferred that new submissions be under one of the two preferred licenses. If you have previously submitted code to RTEMS under a historical license, please grant the project permission to relicense. See https://gitlab.rtems.org/rtems/rtos/rtems/-/issues/3053 for details.

For example templates for what to include in source code and documentation, see *Copyright and License Block* (page 163).

## 13.1 Rationale

RTEMS is intended for use in real-time embedded systems in which the application is statically linked with the operating system and all support libraries. Given this use case, the RTEMS development team evaluated a variety of licenses with with the goal of promoting use while protecting both users and the developers.

Using the GNU General Public License Version 2 (GPLv2) unmodified was considered but discarded because the GPL can only be linked statically with other GPL code. Put simply, linking your application code statically with GPL code would cause your code to become GPL code. This would force both licensing and redistribution requirements onto RTEMS users. This was completely unacceptable.

The GNU Lesser General Public License Version 2 (LGPLv2) was also considered and deemed to not be a suitable license for RTEMS. This is because it either requires use of a shared library that can be re-linked, or release of the linked (application) code. This would require an RTEMS-based embedded system to provide a "relinking kit." Again, this license would force an unacceptable requirement on RTEMS users and deemed unacceptable.

Newer versions of the GPL (i.e. version 3) are completely unsuitable for embedded systems due to the additions which add further restrictions on end user applications.

The historical RTEMS License is a modified version of the GPL version 2 that includes an exception to permit including headers and linking against RTEMS object files statically. This was based on the license used by GCC language runtime libraries at that time. This license allows the static linking of RTEMS with applications without forcing obligations and restrictions on users.

A problem for RTEMS is there are no copyleft licenses that are compatible with the deployment model of RTEMS. Thus, RTEMS Project has to reject any code that uses the GPL or LGPL, even though RTEMS has historically appeared to use the GPL itself – but with the exception for static linking, and also because an upstream GPL version 2 project could at any time switch to GPL version 3 and become totally unusable. In practice, RTEMS can only accept original code contributed under the RTEMS License and code that has a permissive license.

As stated above, the RTEMS Project has defined its preferred licenses. These allow generation of documentation and software from specification as well as allow end users to statically link with RTEMS and not incur obligations.

In some cases, RTEMS includes software from third-party projects. In those cases, the license is carefully evaluated to meet the project licensing goals. The RTEMS Project can only include software under licenses which follow these guidelines:

- 2- and 3-clause BSD, MIT, and other OSI-approved non-copyleft licenses that permit statically linking with the code of different licenses are acceptable.
- The historical RTEMS License is acceptable for software already in the tree. This software is being relicensed to BSD-2-Clause, rewritten, or removed.
- GPL licensed code is NOT acceptable, neither is LGPL.
- Software which is dual-licensed in a manner which prevents free use in commercial applications is not acceptable.
- Advertising obligations are not acceptable.
- Some license restrictions may be permissible. These will be considered on a case-by-case basis. See below for a list of such restrictions.

In practice, these guidelines are not hard to follow. Critically, they protect the freedom of the RTEMS source code and that of end users to select the license and distribution terms they prefer for their RTEMS-based application.

## 13.2 License restrictions

- Apache License 2.0 in section 4 (b) requires modified files to carry prominent notice about performed modification. In case you modify such file and the notice is not there yet you are required to put notice below at the top of the modified file. If the notice is already there you don't need to add it second time. The notice should look:

```
/*
 * The file was modified by RTEMS contributors.
 */
```

> **Warning**
>
> Do not import any project or files covered by the Apache License 2.0 into the RTEMS project source tree without discussing first with developers on the mailing list! Handling of Apache License 2.0 projects is a bit sensitive manner and RTEMS project is not prepared to handle one kind of those projects yet. E.g. the projects with NOTICE file present in the source tree.

CHAPTER

**FOURTEEN**

# APPENDIX: CORE QUALIFICATION ARTIFACTS/DOCUMENTS

An effort at NASA has been performed to suggest a core set of artifacts (as defined by **BOTH** NASA NPR 7150.2B and DO-178B) that can be utilized by a mission as a baselined starting point for "pre-qualification" for (open-source) software that is intended to be utilized for flight purposes. This effort analyzed the overlap between NPR 7150.2B and DO-178B and highlighted a core set of artifacts to serve as a starting point for any open-source project. These artifacts were also cross-referenced with similar activities for other NASA flight software qualification efforts, such as the open-source Core Flight System (cFS). Along with the specific artifact, the intent of the artifact was also captured; in some cases open-source projects, such as RTEMS, are already meeting the intent of the artifacts with information simply needing organized and formalized. The table below lists the general category, artifact name, and its intent. Please note that this table does **NOT** represent all the required artifacts for qualification per the standards; instead, this table represents a subset of the most basic/core artifacts that form a strong foundation for a software engineering qualification effort.

Table 14.1: Table 1. Core Qualification Artifacts

| Category | Artifact | Intent |
|---|---|---|
| Requirements | Software Requirements Specification (SRS) | The project shall document the software requirements. The project shall collect and manage changes to the software requirements. The project shall identify, initiate corrective actions, and track until closure inconsistencies among requirements, project plans, and software products. |
| | Requirements Test and Traceability Matrix | The project shall perform, document, and maintain bidirectional traceability between the software requirement and the higher-level requirement. |
| | Validation | The project shall perform validation to ensure that the software will perform as intended in the customer environment. |
| Design and Implementation | Software Development or Management Plan | A plan for how you will develop the software that you are intent upon developing and delivering. The Software Development Plan includes the objectives, standards and life cycle(s) to be used in the software development process. This plan should include: Standards: Identification of the Software Requirements Standards, Software Design Standards, and Software Code Standards for the project. |
| | Software Configuration Management Plan | To identify and control major software changes, ensure that change is being properly implemented, and report changes to any other personnel or clients who may have an interest. |
| | Implementation | The project shall implement the software design into software code. Executable Code to applicable tested software. |
| | Coding Standards Report | The project shall ensure that software coding methods, standards, and/or criteria are adhered to and verified. |
| | Version Description Document (VDD) | The project shall provide a Software Version Description document for each software release. |
| Testing and Software Assurance Activities | Software Test Plan | Document describing the testing scope and activities. |
| | Software Assurance/Testing Procedures | To define the techniques, procedures, and methodologies that will be used. |
| | Software Change Report / Problem Report | The project shall regularly hold reviews of software activities, status, and results with the project stakeholders and track issues to resolution. |
| | Software Schedule | Milestones have schedule and schedule is updated accordingly. |
| | Software Test Report | The project shall record, address, and track to closure the results of software verification activities. |

In an effort to remain lightweight and sustainable for open-source projects, Table 1 above was condensed into a single artifact outline that encompasses the artifacts' intents. The idea is that this living qualification document will reside under RTEMS source control and be updated with additional detail accordingly. The artifact outline is as follows:

CHAPTER

**FIFTEEN**

# APPENDIX: RTEMS FORMAL MODEL GUIDE

This appendix covers the various formal models of RTEMS that are currently in existence. It serves two purposes: one is to provide detailed documentation of each model, while the other is provide a guide into how to go about developing and deploying such models.

The general approach followed here is to start by looking at the API documentation and identifying the key data-structures and function prototypes. These are then modelled appropriately in Promela. Then, general behavior patterns of interest are identified, and the Promela model is extended to provide those patterns. A key aspect here is exploiting the fact that Promela allows non-deterministic choices to be specified, which gives the effect of producing arbitrary orderings of model behavior. All of this leads to a situation were the SPIN model-checker can effectively generate scenarios for all possible interleavings. The final stage is mapping those scenarios to RTEMS C test code, which has two parts: generating machine-readable output from SPIN, and developing the refinement mapping from that output to C test code.

Some familiarity is assumed here with the Software Test Framework section in this document.

The following models are included in the directory `formal/promela/models/` at the top-level in `rtems-central`:

**Chains API (`chains/`)**
Models the unprotected chain append and get API calls in the Classic Chains API Guide. This was an early model to develop the basic methodology.

**Events Manager (`events/`)**
Models the behaviour of all the API calls in the Classic Events Manager API Guide. This had to tackle real concurrency and deal with multiple CPUs and priority issues.

**Barrier Manager (`barriers/`)**
Models the behaviour of all the API calls in then Classic Barrier Manager API.

**Message Manager (`messages/`)**
Models the create, send and receive API calls in the Classic Message Manager API.

At the end of this guide is a section that discusses various issues that should be tackled in future work.

## 15.1 Testing Chains

Documentation: Chains section in the RTEMS Classic API Guide.

Model Directory: `formal/promela/models/chains`.

Model Name: `chains-api-model`.

The Chains API provides a doubly-linked list data-structure, optimised for fast operations in an SMP setting. It was used as proof of concept exercise, and focussed on just two API calls: `rtems-chain-append-unprotected` and `rtems-chain-get-unprotected` (hereinafter just `append` and `get`).

### 15.1.1 API Model

File: `chains-api-model.pml`

While smart code optimization techniques are very important for RTEMS code, the focus when constructing formal models is on functional correctness, not performance. What is required is the simplest, most obviously correct model.

The `append` operation adds new nodes on the end of the list, while `get` removes and returns the node at the start of the list. The Chains API has many other operations that can add/remove nodes at either end, or somewhere in the middle, but these are considered out of scope.

#### 15.1.1.1 Data Structures

There are no pointers in Promela, so we have to use arrays, with array indices modelling pointers. With just `append` and `get`, an array can be used to implement a collection of nodes in memory. A `Node` type is defined that has next and previous indices, plus an item payload. Access to the node list is via a special control node with head and tail pointers. In the model, an explicit size value is added to this control node, to allow the writing of properties about chain length, and to prevent array out-of-bound errors in the model itself. We assume a single `chain`, with list node storage statically allocated in `memory`.

```
#define PTR_SIZE 3
#define MEM_SIZE 8

typedef Node {
  unsigned nxt  : PTR_SIZE
; unsigned prv  : PTR_SIZE
; byte     itm
}
Node memory[MEM_SIZE] ;

typedef Control {
  unsigned head : PTR_SIZE;
  unsigned tail : PTR_SIZE;
  unsigned size : PTR_SIZE
}
Control chain ;
```

While there are 8 memory elements, element 0 is inaccessible, as the index 0 is treated like a `NULL` pointer.

#### 15.1.1.2 Function Calls

The RTEMS prototype for append is:

```
void rtems_chain_append_unprotected(
    rtems_chain_control *the_chain,
    rtems_chain_node    *the_node
);
```

Its implementation starts by checking that the node to be appended is "off chain", before performing the append. The model is designed to satisfy this property so the check is not modelled. Also, the Chains documentation is not clear about certain error cases. As this is a proof of concept exercise, these details are not modelled.

A Promela inline definition append models the desired behavior, simulating C pointers with array addresses. Here ch is the chain argument, while np is a node index. The model starts by checking that the node pointer is not NULL, and that there is room in memory for another node. These are to ensure that the model does not have any runtime errors. Doing a standard model-check of this model finds no errors, which indicates that those assertions are never false.

```
inline append(ch,np) {
  assert(np!=0); assert(ch.size < (MEM_SIZE-1));
  if
  :: (ch.head == 0) -> ch.head = np; ch.tail = np; ch.size = 1;
                       memory[np].nxt = 0; memory[np].prv = 0;
  :: (ch.head != 0) -> memory[ch.tail].nxt = np; memory[np].prv = ch.tail;
                       ch.tail = np; ch.size = ch.size + 1;
  fi
}
```

The RTEMS prototype for get is:

```
rtems_chain_node *rtems_chain_get_unprotected(
  rtems_chain_control *the_chain
);
```

It returns a pointer to the node, with NULL returned if the chain is empty.

Promela inlines involve textual substitution, so the concept of returning a value makes no sense. For get, the model is that of a statement that assigns the return value to a variable. Both the function argument and return variable name are passed as parameters:

```
/* np = get(ch); */
inline get(ch,np) {
  np = ch.head ;
  if
    :: (np != 0) ->
        ch.head = memory[np].nxt;
        ch.size = ch.size - 1;
        // memory[np].nxt = 0
    :: (np == 0) -> skip
  fi
  if
```

(continues on next page)

(continued from previous page)

```
    :: (ch.head == 0) -> ch.tail = 0
    :: (ch.head != 0) -> skip
  fi
}
```

### 15.1.2 Behavior patterns

File: `chains-api-model.pml`

A key feature of using a modelling language like Promela is that it has both explicit and implicit non-determinism. This can be exploited so that SPIN will find all possible interleavings of behavior.

The Chains API model consists of six processes, three which perform `append`, and three that perform `get`, waiting if the chain is empty. This model relies on implicit non-determinism, in that the SPIN scheduler can choose and switch between any unblocked process at any point. There is no explicit non-determinism in this model.

Promela process `doAppend` takes node index `addr` and a value `val` as parameters. It puts `val` into the node indexed by `addr`, then calls `append`, and terminates. It is all made atomic to avoid unnecessary internal interleaving of operations because unprotected versions of API calls should only be used when interrupts are disabled.

```
proctype doAppend(int addr; int val) {
  atomic{ memory[addr].itm = val;
          append(chain,addr); } ;
}
```

The `doNonNullGet` process waits for the chain to be non-empty before attempting to `get` an element. The first statement inside the atomic construct is an expression, as a statements, that blocks while it evaluates to zero. That only happens if `head` is in fact zero. The model also has an assertion that checks that a non-null node is returned.

```
proctype doNonNullGet() {
  atomic{
    chain.head != 0;
    get(chain,nptr);
    assert(nptr != 0);
  } ;
}
```

All processes terminate after they have performed their (sole) action.

The top-level of a Promela model is an initial process declared by the `init` construct. This initializes the chain as empty and then runs all six processes concurrently. It then uses the special `_nr_pr` variable to wait for all six processes to terminate. A final assertion checks that the chain is empty.

```
init {
  pid nr;
  chain.head = 0; chain.tail = 0; chain.size = 0 ;
  nr = _nr_pr;  // assignment, sets `nr` to current number of procs
```

(continues on next page)

(continued from previous page)

```
  run doAppend(6,21);
  run doAppend(3,22);
  run doAppend(4,23);
  run doNonNullGet();
  run doNonNullGet();
  run doNonNullGet();
  nr == _nr_pr; // expression, waits until number of procs equals `nr`
  assert (chain.size == 0);
}
```

Simulation of this model will show some execution sequence in which the appends happen in a random order, and the gets also occur in a random order, whenever the chain is not empty. All assertions are always satisfied, including the last one above. Model checking this model explores all possible interleavings and reports no errors of any kind. In particular, when the model reaches the last assert statement, the chain size is always zero.

SPIN uses the C pre-processor, and generates the model as a C program. This model has a simple flow of control: basically execute each process once in an almost arbitrary order, assert that the chain is empty, and terminate. Test generation here just requires the negation of the final assertion to get all possible interleavings. The special C pre-processor definition `TEST_GEN` is used to switch between the two uses of the model. The last line above is replaced by:

```
#ifdef TEST_GEN
  assert (chain.size != 0);
#else
  assert (chain.size == 0);
#endif
```

A test generation run can then be invoked by passing in `-DTEST_GEN` as a command-line argument.

### 15.1.3 Annotations

File: `chains-api-model.pml`

The model needs to have `printf` statements added to generation the annotations used to perform the test generation.

This model wraps each of six API calls in its own process, so that model checking can generate all feasible interleavings. However, the plan for the test code is that it will be just one RTEMS Task, that executes all the API calls in the order determined by the scenario under consideration. All the annotations in this model specify `0` as the Promela process identifier.

#### 15.1.3.1 Data Structures

Annotations have to be provided for any variable or datastructure declarations that will need to have corresponding code in the test program. These have to be printed out as the model starts to run. For this model, the `MAX_SIZE` parameter is important, as are the variables `memory`, `nptr`, and `chain`:

```
printf("@@@ 0 NAME Chain_AutoGen\n")
printf("@@@ 0 DEF MAX_SIZE 8\n");
```

(continues on next page)

(continued from previous page)

```
printf("@@@ 0 DCLARRAY Node memory MAX_SIZE\n");
printf("@@@ 0 DECL unsigned nptr NULL\n")
printf("@@@ 0 DECL Control chain\n");
```

At this point, a parameter-free initialization annotation is issued. This should be refined to C code that initializes the above variables.

```
printf("@@@INIT\n");
```

#### 15.1.3.2 Function Calls

For append, two forms of annotation are produced. One uses the `CALL` format to report the function being called along with its arguments. The other form reports the resulting contents of the chain.

```
proctype doAppend(int addr; int val) {
  atomic{ memory[addr].itm = val; append(chain,addr);
          printf("@@@ 0 CALL append %d %d\n",val,addr);
          show_chain();
        } ;
}
```

The statement `show_chain()` is an inline function that prints the contents of the chain after append returns. The resulting output is multi-line, starting with `@@@ 0 SEQ chain`, ending with `@@@ 0 END chain`, and with entries in between of the form `@@@ 0 SCALAR _ val` displaying chain elements, line by line.

Something similar is done for get, with the addition of a third annotation `show_node()` that shows the node that was got:

```
proctype doNonNullGet() {
  atomic{
    chain.head != 0;
    get(chain,nptr);
    printf("@@@ 0 CALL getNonNull %d\n",nptr);
    show_chain();
    assert(nptr != 0);
    show_node();
  } ;
}
```

The statement `show_node()` is defined as follows:

```
inline show_node (){
  atomic{
    printf("@@@ 0 PTR nptr %d\n",nptr);
    if
    :: nptr -> printf("@@@ 0 STRUCT nptr\n");
               printf("@@@ 0 SCALAR itm %d\n", memory[nptr].itm);
               printf("@@@ 0 END nptr\n")
    :: else -> skip
```

(continues on next page)

(continued from previous page)

```
    fi
  }
}
```

It prints out the value of `nptr`, which is an array index. If it is not zero, it prints out some details of the indexed node structure.

Annotations are also added to the `init` process to show the chain and node.

```
chain.head = 0; chain.tail = 0; chain.size = 0;
show_chain();
show_node();
```

### 15.1.4 Refinement

Files:

`chains-api-model-rfn.yml`

`chains-api-model-pre.h`

`tr-chains-api-model.c`

Model annotations are converted to C test code using a YAML file that maps single names to test code snippets into which parameters can be substituted. Parameters are numbered from zero, and the `n` th parameter will be substituted wherever `{n}` occurs in the snippet.

Refinement is more than just the above mapping from annotations to code. Some of this code will refer to C variables, structures, and functions that are needed to support the test. Some of these are declared `chains-api-model-pre.h` and implemented in `tr-chains-api-model.c`.

#### 15.1.4.1 Data Structures

The initialization generates one each of `NAME`, `DEF`, `DCLARRAY`, and `INIT` annotations, and two `DECL` annotations.

The `DEF` entry is currently not looked up as it is automatically converted to a `#define`.

The `NAME` annotation is used to declare the test case name, which is stored in the global variable `rtems_test_name`. The current refinement entry is:

```
NAME: |
  const char rtems_test_name[] = "Model_Chain_API";
```

In this case, the name is fixed and ignores what is declared in the model.

The `DCLARRAY Node memory MAX_SIZE` annotation looks up `memory_DCL` in the refinement file, passing in `memory` and `MAX_SIZE` as arguments. The entry in the refinement file is:

```
memory_DCL: item {0}[{1}];
```

Here `item` is the type of the chains nodes used in the test code. It is declared in `chains-api-model.pre.h` as:

```
typedef struct item
{
    rtems_chain_node node;
    int               val;
} item;
```

Substituting the arguments gives:

```
item memory[MAX_SIZE];
```

The two DECL annotations have two or three parameters. The first is the type, the second is the variable name, and the optional third is an initial value. The lookup key is the variable name with _DCL added on. In the refinement file, the entry only provides the C type, and the other parts of the declaration are added in. The entries are:

```
nptr_DCL: item *
chain_DCL: rtems_chain_control
```

Annotation DECL unsigned nptr NULL results in:

```
item * nptr = NULL ;
```

Annotation DECL Control chain results in:

```
rtems_chain_control chain ;
```

The INIT annotation is looked up as INIT itself. It should be mapped to code that does all necessary initialization. The refinement entry for chains is:

```
INIT: rtems_chain_initialize_empty( &chain );
```

In addition to all the above dealing with declarations and initialization, there are the annotations, already described above, that are used to display composite values, such as structure contents, and chain contents.

In the model, all accesses to individual chain nodes are via index nptr, which occurs in two types of annotations, PTR and STRUCT. The PTR annotation is refined by looking up nptr_PTR with the value of nptr as the sole argument. The refinement entry is:

```
nptr_PTR: |
  T_eq_ptr( nptr, NULL );
  T_eq_ptr( nptr, &memory[{0}] );
```

The first line is used if the value of nptr is zero, otherwise the second line is used.

The use of STRUCT requires three annotation lines in a row, *e.g.*:

```
@@@ 0 STRUCT nptr
@@@ 0 SCALAR itm 21
@@@ 0 END nptr
```

The STRUCT and END annotations do not generate any code, but open and close a scope in which nptr is noted as the "name" of the struct. The SCALAR annotation is used to observe simple

values such as numbers or booleans. However, within a STRUCT it belongs to a C struct, so the relevant field needs to be used to access the value. Within this scope, the scalar variable itm is looked up as a field name, by searching for itm_FSCALAR with arguments``nptr`` and 21, which returns the entry:

```
itm_FSCALAR:   T_eq_int( {0}->val, {1} );
```

This gets turned into the following test:

```
T_eq_int( nptr->val, 21 );
```

A similar idea is used to test the contents of a chain. The annotations produced start with a SEQ annotation, followed by a SCALAR annotation for each item in the chain, and then ending with an END annotation. Again, there is a scope defined where the SEQ argument is the "name" of the sequence. The SCALAR entries have no name here (indicated by _), and their values are accumulated in a string, separated by spaces. Test code generation is triggered this time by the END annotation, that looks up the "name" with _SEQ appended, and the accumulated string as an argument. The corresponding entry for chain sequences is:

```
chain_SEQ: |
  show_chain( &chain, ctx->buffer );
  T_eq_str( ctx->buffer, "{0} 0" );
```

So, the following chain annotation sequence:

```
@@@ 0 SEQ chain
@@@ 0 SCALAR _ 21
@@@ 0 SCALAR _ 22
@@@ 0 END chain
```

becomes the following C code:

```
show_chain( &chain, ctx->buffer );
T_eq_str( ctx->buffer, " 21 22 0" );
```

C function show_chain() is defined in tr-chains-api-model.c and generates a string with exactly the same format as that produced above. These are then compared for equality.

**Note**

The Promela/SPIN model checker's prime focus is modelling and verifying concurrency related properties. It is not intended for verifying sequential code or data transformations. This is why some of the STRUCT/SEQ material here is so clumsy. It plays virtually no role in the other models.

#### 15.1.4.2 Function Calls

For `@@@ 0 CALL append 22 3` lookup append to get

```
memory[{1}].val = {0};
rtems_chain_append_unprotected( &chain, (rtems_chain_node*)&memory[{1}] );
```

Substitute `22` and `3` in to produce

```
memory[3].val = 22;
rtems_chain_append_unprotected( &chain, (rtems_chain_node*)&memory[3] );
```

For `@@@ 0 CALL getNonNull 3` lookup `getNonNull` to obtain

```
nptr = get_item( &chain );
T_eq_ptr( nptr, &memory[{0}] );
```

Function `get_item()` is defined in `tc-chains-api-model.c` and calls `rtems_chain_get_unprotected()`. Substitute `3` to produce:

```
nptr = get_item( &chain );
T_eq_ptr( nptr, &memory[3] );
```

## 15.2 Testing Concurrent Managers

All the other models are of Managers that provide some form of communication between multiple RTEMS Tasks. This introduces a number of issues regarding the timing and control of tasks, particularly when developing *reproducible* tests, where the sequencing of behavior is the same every time the test runs. The tests are generated by following the schemes already in use for regular RTEMS handwritten tests. In particular the pre-existing tests for Send and Receive in the Event Manager where used as a guide.

### 15.2.1 Testing Strategy

The tests are organized as follows:

1. A designated Task, the Runner, is responsible for initializing, coordinating and tearing down a test run. Coordination involves starting other Worker Tasks that perform tests, and waiting for them to complete. It may also run some tests itself.

1) One or more Worker Tasks are used to perform test actions.

1. Each RTEMS Task (Runner/Worker) is modelled by one Promela process.

1) Simple Binary Semaphores are used to coordinate all the tasks to ensure that the interleaving is always the same (See Semaphore Manager section in Classic API Manual).

1. Two other Promela processes are required: One, called `Clock()` is used to model timing and timeouts; The other, called `System()` models relevant behavior of the RTEMS scheduler.

### 15.2.2 Model Structure

All the models developed so far are based on this framework. The structure of these models takes the following form:

**Constant Declarations**
Mainly `#defines` that define important constants.

**Datastructure Definitions**
Promela `typedefs` that describe key forms of data. In particular there is a type `Task` that models RTEMS Tasks. The Simple Binary Semaphores are modelled as boolean variables.

**Global Variable Declarations**
Typically these are arrays of data-structures, representing objects such as tasks or semaphores.

**Supporting Models**
These are `inline` definitions that capture common behavior. In all models this includes `Obtain()` and `Release()` to model semaphores, and `waitUntilReady()` that models a blocked task waiting to be unblocked. Included here are other definitions specific to the particular Manager being modelled.

**API Models**
These are `inline` definitions that model the behavior of each API call. All behavior must be modelled, including bad calls that (should) result in an error code being returned. The parameter lists used in the Promela models will differ from those of the actual API calls. Consider a hypothetical RTEMS API call:

```
rc = rtems_some_api(arg1,arg2,...,argN);
```

One reason, common to all calls, is that the `inline` construct has no concept of returning a value, so a variable parameter has to be added to represent it, and it has to be ensured the appropriate return code is assigned to it.

```
inline some_api(arg1,arg2,...,argN,rc) {
  ...
  rc = RC_some_code
}
```

Another reason is that some RTEMS types encode a number of different concepts in a single machine word. The most notable of these is the `rtems_options` type, that specifies various options, usually for calls that may block. In some models, some options are modelled individually, for clarity. So the API model may have two or more parameters where the RTEMS call has one.

```
inline some_api(arg1,arg2feat1,arg2feat2,...,argN,rc) {
  ...
  rc = RC_some_code
}
```

The refinement of this will pass the multiple feature arguments to a C function that will assemble the single RTEMS argument.

A third reason is that sometimes it is important to also provide the process id of the *calling* task. This can be important where priority and preemption are involved.

**Scenario Generation**
A Testsuite that exercises *all* the API code is highly desirable. This requires running tests that explore a wide range of scenarios, normally devised by hand when writing a testsuite. With Promela/SPIN, the model-checker can generate all of these simplify as a result of its exhaustive search of the model. In practice, scenarios fall into a number of high-level categories, so the first step is make a random choice of such a category. Within a category there may be further choices to be done. A collection of global scenario variables are used to records the choices made. This is all managed by inline `chooseScenario()`.

**RTEMS Test Task Modelling**
This is a series of Promela `proctypes`, one for the Runner Task, and one for each of the Worker Tasks. Their behavior is controlled by the global scenario variables.

**System Modelling**
These are Promela processes that model relevant underlying RTEMS behavior, such as the scheduler (`System()`) and timers (`Clock()`).

**Model Main Process**
Called `init`, this initialises variables, invokes `chooseScenario()`, runs all the processes, waits for them to terminate, and then terminates itself.

The Promela models developed so far for these Managers always terminate. The last few lines of each are of the form:

```
#ifdef TEST_GEN
  assert(false);
#endif
```

If these models are run in the usual way (simulation or verification), then a correct verified model is observed. If `-DTEST_GEN` is provided as the first command-line argument, in verification mode configured to find *all* counterexamples, then all the possible (correct) behaviours of the system will be generated.

### 15.2.3 Transforming Model Behavior to C Code

As described earlier, `printf` statements are used to produce easy to parse output that makes model events and outcomes easy to identify and process. The YAML file used to define the refinement from model to code provides a way of looking up an observation with arguments, and then obtaining a C template that can be populated with those arguments.

This refinement is a bridge between two distinct worlds:

**Model World:**
where the key focus is on correctness and clarity.

**Code World:**
where clarity is often sacrificed for efficiency.

This means that the model-to-code relationship need not be a simple one-to-one mapping. This has already been alluded to above when the need for extra parameters in API call models was discussed. It can also be helpful when the model is better treating various attributes separately, while the code handles those attributes packed into machine words.

It is always reasonable to add test support code to the C test sources, and this can include C functions that map distinct attribute values down into some compact merged representation.

## 15.3 Testing the Event Manager

Documentation: Event Manager section in the RTEMS Classic API Guide.

Model Directory: `formal/promela/models/events`.

Model Name: `event-mgr-model`.

The Event Manager allows tasks to send events to, and receive events from, other tasks. From the perspective of the Event Manager, events are just uninterpreted numbers in the range 0...31, encoded as a 32-bit bitset.

**`rtems_event_send(id,event_in)`**
allows a task to send a bitset to a designated task

**`rtems_event_receive(event_in,option_set,ticks,event_out)`**
allows a task to specify a desired bitset with options on what to do if it is not present.

Most of the requirements are pretty straightforward, but two were a little more complex, and drove the more complex parts of the modelling.

1. If a task was blocked waiting to receive events, and a lower priority task then sent the events that would wake that blocked task, then the sending task would be immediately preempted by the receiver task.
2. There was a requirement that explicitly discussed the situation where the two tasks involved were running on different processors.

A preliminary incomplete model of the Event Manager was originally developed by the consortium early in the project. The model was then completed during the rest of the activity by a Masters student: [Jen21]. They also developed the first iteration of the `testbuilder` program.

### 15.3.1 API Model

File: `event-mgr-model.pml`

The RTEMS Event set contains 32 values, but in the model limits this to just four, which is enough for test purposes. Some inline definitions are provided to encode (`events`), display (`printevents`), and subtract (`setminus`) events.

The `rtems_option_set` is simplifiedto just two relevant bits: the timeout setting (`Wait`, `NoWait`), and how much of the desired event set will satisfy the receiver (`All`, `Any`). These are passed in as two separate arguments to the model of the receive call.

#### 15.3.1.1 Event Send

An RTEMS call `rc = rtems_event_send(tid,evts)` is modelled by an inline of the form:

```
1 event_send(self,tid,evts,rc)
```

**The four arguments are:**

**`self`**
id of process modelling the task/IDR making call.

**`tid`**
id of process modelling the target task of the call.

**evts**
event set being sent.

**rc**
updated with the return code when the send completes.

The main complication in the otherwise straightforward model is the requirement to preempt under certain circumstances.

```
inline event_send(self,tid,evts,rc) {
  atomic{
    if
    ::  tid >= BAD_ID -> rc = RC_InvId
    ::  tid < BAD_ID ->
        tasks[tid].pending = tasks[tid].pending | evts
        // at this point, have we woken the target task?
        unsigned got : NO_OF_EVENTS;
        bool sat;
        satisfied(tasks[tid],got,sat);
        if
        ::  sat ->
            tasks[tid].state = Ready;
            printf("@@@ %d STATE %d Ready\n",_pid,tid)
            preemptIfRequired(self,tid) ;
            // tasks[self].state may now be OtherWait !
            waitUntilReady(self);
        ::  else -> skip
        fi
        rc = RC_OK;
    fi
  }
}
```

Three inline abstractions are used:

**satisfied(task,out,sat)**
updates `out` with the wanted events received so far, and then checks if a receive has been satisfied. It updates its `sat` argument to reflect the check outcome.

**preemptIfRequired(self,tid)**
forces the sending process to enter the `OtherWait`, if circumstances require it.

**waitUntilReady(self)**
basically waits for the process state to become `Ready`.

#### 15.3.1.2 Event Receive

An RTEMS call `rc = rtems_event_receive(evts,opts,interval,out)` is modelled by an inline of the form:

```
event_receive(self,evts,wait,wantall,interval,out,rc)
```

**The seven arguments are:**

**self**
id of process modelling the task making call

**evts**
input event set

**wait**
true if receive should wait

**what**
all, or some?

**interval**
wait interval (0 waits forever)

**out**
pointer to location for satisfying events when the receive completes.

**rc**
updated with the return code when the receive completes.

There is a small complication, in that there are distinct variables in the model for receiver options that are combined into a single RTEMS option set. The actual calling sequence in C test code will be:

```
opts = mergeopts(wait,wantall);
rc = rtems_event_receive(evts,opts,interval,out);
```

Here `mergeopts` is a C function defined in the C Preamble.

```
inline event_receive(self,evts,wait,wantall,interval,out,rc){
  atomic{
    printf("@@@ %d LOG pending[%d] = ",_pid,self);
    printevents(tasks[self].pending); nl();
    tasks[self].wanted = evts;
    tasks[self].all = wantall
    if
    ::  out == 0 ->
        printf("@@@ %d LOG Receive NULL out.\n",_pid);
        rc = RC_InvAddr ;
    ::  evts == EVTS_PENDING ->
        printf("@@@ %d LOG Receive Pending.\n",_pid);
        recout[out] = tasks[self].pending;
        rc = RC_OK
    ::  else ->
        bool sat;
        retry:  satisfied(tasks[self],recout[out],sat);
        if
        ::  sat ->
            printf("@@@ %d LOG Receive Satisfied!\n",_pid);
            setminus(tasks[self].pending,tasks[self].pending,recout[out]);
            printf("@@@ %d LOG pending'[%d] = ",_pid,self);
            printevents(tasks[self].pending); nl();
            rc = RC_OK;
```

(continues on next page)

(continued from previous page)

```
25          ::  !sat && !wait ->
26              printf("@@@ %d LOG Receive Not Satisfied (no wait)\n",_pid);
27              rc = RC_Unsat;
28          ::  !sat && wait && interval > 0 ->
29              printf("@@@ %d LOG Receive Not Satisfied (timeout %d)\n",_pid,
↪interval);
30              tasks[self].ticks = interval;
31              tasks[self].tout = false;
32              tasks[self].state = TimeWait;
33              printf("@@@ %d STATE %d TimeWait %d\n",_pid,self,interval)
34              waitUntilReady(self);
35              if
36              ::  tasks[self].tout  ->  rc = RC_Timeout
37              ::  else              ->  goto retry
38              fi
39          ::  else -> // !sat && wait && interval <= 0
40              printf("@@@ %d LOG Receive Not Satisfied (wait).\n",_pid);
41              tasks[self].state = EventWait;
42              printf("@@@ %d STATE %d EventWait\n",_pid,self)
43              if
44              :: sendTwice && !sentFirst -> Released(sendSema);
45              :: else
46              fi
47              waitUntilReady(self);
48              goto retry
49          fi
50      fi
51      printf("@@@ %d LOG pending'[%d] = ",_pid,self);
52      printevents(tasks[self].pending); nl();
53    }
54  }
```

Here `satisfied()` and `waitUntilReady()` are also used.

### 15.3.2 Behaviour Patterns

File: `event-mgr-model.pml`

The Event Manager model consists of five Promela processes:

**`init`**
: The first top-level Promela process that performs initialisation, starts the other processes, waits for them to terminate, and finishes.

**`System`**
: A Promela process that models the behaviour of the operating system, in particular that of the scheduler.

**`Clock`**
: A Promela process used to facilitate modelling timeouts.

**`Receiver`**
: The Promela process modelling the test Runner, that also invokes the receive API call.

**Sender**
A Promela process modelling a singe test Worker that invokes the send API call.

Two simple binary semaphores are used to synchronise the tasks.

The Task model only looks at an abstracted version of RTEMS Task states:

**Zombie**
used to model a task that has just terminated. It can only be deleted.

**Ready**
same as the RTEMS notion of `Ready`.

**EventWait**
is `Blocked` inside a call of `event_receive()` with no timeout.

**TimeWait**
is `Blocked` inside a call of `event_receive()` with a timeout.

**OtherWait**
is `Blocked` for some other reason, which arises in this model when a sender gets pre-empted by a higher priority receiver it has just satisfied.

Tasks are represented using a datastructure array. As array indices are proxies here for C pointers, the zeroth array entry is always unused, as index value 0 is used to model a NULL C pointer.

```
typedef Task {
  byte nodeid; // So we can spot remote calls
  byte pmlid; // Promela process id
  mtype state ; // {Ready,EventWait,TickWait,OtherWait}
  bool preemptable ;
  byte prio ; // lower number is higher priority
  int ticks; //
  bool tout; // true if woken by a timeout
  unsigned wanted  : NO_OF_EVENTS ; // EvtSet, those expected by receiver
  unsigned pending : NO_OF_EVENTS ; // EvtSet, those already received
  bool all; // Do we want All?
};
Task tasks[TASK_MAX]; // tasks[0] models a NULL dereference
```

```
byte sendrc;            // Sender global variable
byte recrc;             // Receiver global variable
byte recout[TASK_MAX] ; // models receive 'out' location.
```

#### 15.3.2.1 Task Scheduling

In order to produce a model that captures real RTEMS Task behaviour, there need to be mechanisms that mimic the behaviour of the scheduler and other activities that can modify the execution state of these Tasks. Given a scenario generated by such a model, synchronisation needs to be added to the generated C code to ensure test has the same execution patterns.

Relevant scheduling behavior is modelled by two inlines that have already been mentioned: `waitUntilReady()` and `preemptIfRequired()`.

For synchronisation, simple boolean semaphores are used, where True means available, and False means the semaphore has been acquired.

```
bool semaphore[SEMA_MAX]; // Semaphore
```

The synchronisation mechanisms are:

**Obtain(sem_id)**
call that waits to obtain semaphore sem_id.

**Release(sem_id)**
call that releases semaphore sem_id

**Released(sem_id)**
simulates ecosystem behaviour that releases sem_id.

The difference between Release and Released is that the first issues a SIGNAL annotation, while the second does not.

#### 15.3.2.2 Scenarios

A number of different scenario schemes were defined that cover various aspects of Event Manager behaviour. Some schemes involve only one task, and are usually used to test error-handling or abnormal situations. Other schemes involve two tasks, with some mixture of event sending and receiving, with varying task priorities.

For example, an event send operation can involve a target identifier that is invalid (BAD_ID), correctly identifies a receiver task (RCV_ID), or is sending events to itself (SEND_ID).

```
typedef SendInputs {
  byte target_id ;
  unsigned send_evts : NO_OF_EVENTS ;
} ;
SendInputs  send_in[MAX_STEPS];
```

An event receive operation will be determined by values for desired events, and the relevant to bits of the option-set parameter.

```
typedef ReceiveInputs {
  unsigned receive_evts : NO_OF_EVENTS ;
  bool will_wait;
  bool everything;
  byte wait_length;
};
ReceiveInputs receive_in[MAX_STEPS];
```

There is a range of global variables that define scenarios for both send and receive. This defines a two-step process for choosing a scenario. The first step is to select a scenario scheme. The possible schemes are defined by the following mtype:

```
mtype = {Send,Receive,SndRcv,RcvSnd,SndRcvSnd,SndPre,MultiCore};
mtype scenario;
```

One of these is chosen by using a conditional where all alternatives are executable, so behaving as a non-deterministic choice of one of them.

```
if
::  scenario = Send;
::  scenario = Receive;
::  scenario = SndRcv;
::  scenario = SndPre;
::  scenario = SndRcvSnd;
::  scenario = MultiCore;
fi
```

Once the value of scenario is chosen, it is used in another conditional to select a non-deterministic choice of the finer details of that scenario.

```
if
::  scenario == Send ->
      doReceive = false;
      sendTarget = BAD_ID;
::  scenario == Receive ->
      doSend = false
      if
      :: rcvWait = false
      :: rcvWait = true; rcvInterval = 4
      :: rcvOut = 0;
      fi
      printf( "@@@ %d LOG sub-senario wait:%d interval:%d, out:%d\n",
              _pid, rcvWait, rcvInterval, rcvOut )
::  scenario == SndRcv ->
      if
      ::  sendEvents = 14; // {1,1,1,0}
      ::  sendEvents = 11; // {1,0,1,1}
      fi
      printf( "@@@ %d LOG sub-senario send-receive events:%d\n",
              _pid, sendEvents )
::  scenario == SndPre ->
      sendPrio = 3;
      sendPreempt = true;
      startSema = rcvSema;
      printf( "@@@ %d LOG sub-senario send-preemptable events:%d\n",
              _pid, sendEvents )
::  scenario == SndRcvSnd ->
      sendEvents1 = 2; // {0,0,1,0}
      sendEvents2 = 8; // {1,0,0,0}
      sendEvents = sendEvents1;
      sendTwice = true;
      printf( "@@@ %d LOG sub-senario send-receive-send events:%d\n",
              _pid, sendEvents )
::  scenario == MultiCore ->
      multicore = true;
      sendCore = 1;
      printf( "@@@ %d LOG sub-senario multicore send-receive events:%d\n",
              _pid, sendEvents )
```

(continues on next page)

(continued from previous page)

```
::  else // go with defaults
fi
```

Ddefault values are defined for all the global scenario variables so that the above code focusses on what differs. The default scenario is a receiver waiting for a sender of the same priority which sends exactly what was requested.

#### 15.3.2.3 Sender Process (Worker Task)

The sender process then uses the scenario configuration to determine its behaviour. A key feature is the way it acquires its semaphore before doing a send, and releases the receiver semaphore when it has just finished sending. Both these semaphores are initialised in the unavailable state.

```
proctype Sender (byte nid, taskid) {

  tasks[taskid].nodeid = nid;
  tasks[taskid].pmlid = _pid;
  tasks[taskid].prio = sendPrio;
  tasks[taskid].preemptable = sendPreempt;
  tasks[taskid].state = Ready;
  printf("@@@ %d TASK Worker\n",_pid);
  if
  :: multicore ->
       // printf("@@@ %d CALL OtherScheduler %d\n", _pid, sendCore);
       printf("@@@ %d CALL SetProcessor %d\n", _pid, sendCore);
  :: else
  fi
  if
  :: sendPrio > rcvPrio -> printf("@@@ %d CALL LowerPriority\n", _pid);
  :: sendPrio == rcvPrio -> printf("@@@ %d CALL EqualPriority\n", _pid);
  :: sendPrio < rcvPrio -> printf("@@@ %d CALL HigherPriority\n", _pid);
  :: else
  fi
repeat:
  Obtain(sendSema);
  if
  :: doSend ->
    if
    :: !sentFirst -> printf("@@@ %d CALL StartLog\n",_pid);
    :: else
    fi
    printf("@@@ %d CALL event_send %d %d %d sendrc\n",_pid,taskid,sendTarget,
↪sendEvents);
    if
    :: sendPreempt && !sentFirst -> printf("@@@ %d CALL CheckPreemption\n",_pid);
    :: !sendPreempt && !sentFirst -> printf("@@@ %d CALL CheckNoPreemption\n",_
↪pid);
    :: else
    fi
```

(continues on next page)

(continued from previous page)

```
    event_send(taskid,sendTarget,sendEvents,sendrc);
    printf("@@@ %d SCALAR sendrc %d\n",_pid,sendrc);
  :: else
  fi
  Release(rcvSema);
  if
  :: sendTwice && !sentFirst ->
     sentFirst = true;
     sendEvents = sendEvents2;
     goto repeat;
  :: else
  fi
  printf("@@@ %d LOG Sender %d finished\n",_pid,taskid);
  tasks[taskid].state = Zombie;
  printf("@@@ %d STATE %d Zombie\n",_pid,taskid)
}
```

#### 15.3.2.4 Receiver Process (Runner Task)

The receiver process uses the scenario configuration to determine its behaviour. It has the responsibility to trigger the start semaphore to allow either itself or the sender to start. The start semaphore corresponds to either the send or receive semaphore, depending on the scenario. The receiver acquires the receive semaphore before proceeding, and releases the send sempahore when done.

```
proctype Receiver (byte nid, taskid) {

  tasks[taskid].nodeid = nid;
  tasks[taskid].pmlid = _pid;
  tasks[taskid].prio = rcvPrio;
  tasks[taskid].preemptable = false;
  tasks[taskid].state = Ready;
  printf("@@@ %d TASK Runner\n",_pid,taskid);
  if
  :: multicore ->
       printf("@@@ %d CALL SetProcessor %d\n", _pid, rcvCore);
  :: else
  fi
  Release(startSema); // make sure stuff starts */
  /* printf("@@@ %d LOG Receiver Task %d running on Node %d\n",_pid,taskid,nid);␣
↪*/
  Obtain(rcvSema);

  // If the receiver is higher priority then it will be running
  // The sender is either blocked waiting for its semaphore
  // or because it is lower priority.
  // A high priority receiver needs to release the sender now, before it
  // gets blocked on its own event receive.
  if
```

(continues on next page)

(continued from previous page)

```
  :: rcvPrio < sendPrio -> Release(sendSema);  // Release send semaphore for preemption
  :: else
  fi
  if
  :: doReceive ->
    printf("@@@ %d SCALAR pending %d %d\n",_pid,taskid,tasks[taskid].pending);
    if
    :: sendTwice && !sentFirst -> Release(sendSema)
    :: else
    fi
    printf("@@@ %d CALL event_receive %d %d %d %d %d recrc\n",
           _pid,rcvEvents,rcvWait,rcvAll,rcvInterval,rcvOut);
              /* (self,  evts,     when,   what,  ticks,      out,   rc) */
    event_receive(taskid,rcvEvents,rcvWait,rcvAll,rcvInterval,rcvOut,recrc);
    printf("@@@ %d SCALAR recrc %d\n",_pid,recrc);
    if
    :: rcvOut > 0 ->
      printf("@@@ %d SCALAR recout %d %d\n",_pid,rcvOut,recout[rcvOut]);
    :: else
    fi
    printf("@@@ %d SCALAR pending %d %d\n",_pid,taskid,tasks[taskid].pending);
  :: else
  fi
  Release(sendSema);
  printf("@@@ %d LOG Receiver %d finished\n",_pid,taskid);
  tasks[taskid].state = Zombie;
  printf("@@@ %d STATE %d Zombie\n",_pid,taskid)
}
```

#### 15.3.2.5 System Process

A process is needed that periodically wakes up blocked processes. This is modelling background behaviour of the system, such as ISRs and scheduling. All tasks are visited in round-robin order (to prevent starvation) and are made ready if waiting on other things. Tasks waiting for events or timeouts are not touched. This terminates when all tasks are zombies.

```
proctype System () {
  byte taskid ;
  bool liveSeen;
  printf("@@@ %d LOG System running...\n",_pid);
  loop:
  taskid = 1;
  liveSeen = false;
  printf("@@@ %d LOG Loop through tasks...\n",_pid);
  atomic {
    printf("@@@ %d LOG Scenario is ",_pid);
    printm(scenario); nl();
```

(continues on next page)

(continued from previous page)

```
  }
  do   // while taskid < TASK_MAX ....
  ::  taskid == TASK_MAX -> break;
  ::  else ->
      atomic {
        printf("@@@ %d LOG Task %d state is ",_pid,taskid);
        printm(tasks[taskid].state); nl()
      }
      if
      :: tasks[taskid].state == Zombie -> taskid++
      :: else ->
         if
         ::  tasks[taskid].state == OtherWait
             -> tasks[taskid].state = Ready
                printf("@@@ %d STATE %d Ready\n",_pid,taskid)
         ::  else -> skip
         fi
         liveSeen = true;
         taskid++
      fi
  od
  printf("@@@ %d LOG ...all visited, live:%d\n",_pid,liveSeen);
  if
  ::  liveSeen -> goto loop
  ::  else
  fi
  printf("@@@ %d LOG All are Zombies, game over.\n",_pid);
  stopclock = true;
}
```

#### 15.3.2.6 Clock Process

A process is needed that handles a clock tick, by decrementing the tick count for tasks waiting on a timeout. Such a task whose ticks become zero is then made `Ready`, and its timer status is flagged as a timeout. This terminates when all tasks are zombies (as signalled by `System()` via `stopclock`).

```
proctype Clock () {
  int tid, tix;
  printf("@@@ %d LOG Clock Started\n",_pid)
  do
  ::  stopclock  -> goto stopped
  ::  !stopclock ->
      printf(" (tick) \n");
      tid = 1;
      do
      ::  tid == TASK_MAX -> break
      ::  else ->
          atomic{
```

(continues on next page)

(continued from previous page)

```
            printf("Clock: tid=%d, state=",tid);
            printm(tasks[tid].state); nl()
          };
          if
          ::  tasks[tid].state == TimeWait ->
              tix = tasks[tid].ticks - 1;
              if
              ::  tix == 0
                  tasks[tid].tout = true
                  tasks[tid].state = Ready
                  printf("@@@ %d STATE %d Ready\n",_pid,tid)
              ::  else
                  tasks[tid].ticks = tix
              fi
          ::  else // state != TimeWait
          fi
          tid = tid + 1
      od
  od
stopped:
  printf("@@@ %d LOG Clock Stopped\n",_pid);
}
```

#### 15.3.2.7 init Process

The initial process outputs annotations for defines and declarations, generates a scenario non-deterministically and then starts the system, clock and send and receive processes running. It then waits for those to complete, and them, if test generation is underway, asserts `false` to trigger a seach for counterexamples:

```
init {
  pid nr;
  printf("@@@ %d NAME Event_Manager_TestGen\n",_pid)
  outputDefines();
  outputDeclarations();
  printf("@@@ %d INIT\n",_pid);
  chooseScenario();
  run System();
  run Clock();
  run Sender(THIS_NODE,SEND_ID);
  run Receiver(THIS_NODE,RCV_ID);
  _nr_pr == 1;
#ifdef TEST_GEN
  assert(false);
#endif
}
```

The information regarding when tasks should wait and/or restart can be obtained by tracking the process identifiers, and noting when they change. The `spin2test` program does this, and also produces separate test code segments for each Promela process.

### 15.3.3 Annotations

File: `event-mgr-model.pml`

Nothing more to say here.

### 15.3.4 Refinement

File: `event-mgr-model-rfn.yml`

The test-code generated here is based on the test-code generated from the specification items used to describe the Event Manager in the main (non-formal) part of the new qualification material.

The relevant specification item is `spec/rtems/event/req/send-receive.yml` found in `rtems-central`. The corresponding C test code is `tr-event-send-receive.c` found in `rtems` at `testsuites/validation`. That automatically generated C code is a single file that uses a set of deeply nested loops to iterate through the scenarios it generates.

The approach here is to generate a stand-alone C code file for each scenario (`tr-event-mgr-model-N.c` for `N` in range 0...8.)

The `TASK` annotations issued by the `Sender` and `Receiver` processes lookup the following refinement entries, to get code that tests that the C code Task does correspond to what is being defined in the model.

```
Runner: |
  checkTaskIs( ctx->runner_id );

Worker: |
  checkTaskIs( ctx->worker_id );
```

The `WAIT` and `SIGNAL` annotations produced by `Obtain()` and `Release()` respectively, are mapped to the corresponding operations on RTEMS semaphores in the test code.

```
code content
SIGNAL: |
  Wakeup( semaphore[{}] );

WAIT: |
  Wait( semaphore[{}] );
```

Some of the `CALL` annotations are used to do more complex test setup involving priorities, or other processors and schedulers. For example:

```
HigherPriority: |
  SetSelfPriority( PRIO_HIGH );
  rtems_task_priority prio;
  rtems_status_code sc;
  sc = rtems_task_set_priority( RTEMS_SELF, RTEMS_CURRENT_PRIORITY, &prio );
  T_rsc_success( sc );
  T_eq_u32( prio, PRIO_HIGH );

SetProcessor: |
  T_ge_u32( rtems_scheduler_get_processor_maximum(), 2 );
```

(continues on next page)

(continued from previous page)

```
  uint32_t processor = {};
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(processor, &cpuset);
```

Some handle more complicated test outcomes, such as observing context-switches:

```
CheckPreemption: |
  log = &ctx->thread_switch_log;
  T_eq_sz( log->header.recorded, 2 );
  T_eq_u32( log->events[ 0 ].heir, ctx->runner_id );
  T_eq_u32( log->events[ 1 ].heir, ctx->worker_id );
```

Most of the other refinement entries are similar to those described above for the Chains API.

## 15.4 Testing the Barrier Mananger

Documentation: Barrier Manager section in the RTEMS Classic API Guide.

Model Directory: `formal/promela/models/barriers`.

Model Name: `barrier-mgr-model`.

The Barrier Manager is used to arrange for a number of tasks to wait on a designated barrier object, until either another task releases them, or a given number of tasks are waiting, at which point they are all released.

All five directives were modelled:

- `rtems_barrier_create()`
- `rtems_barrier_ident()`
- `rtems_barrier_delete()`
- `rtems_barrier_wait()`
- `rtems_barrier_release()`

Barriers can be manual (released only by a call to `..release()`), or automatic (released by the call to `..wait()` that results in a wait count limit being reached.) There is no notion of queuing in this manager, only waiting for a barrier to be released.

This model was developed by a Masters student [Jaskuc22], using the Event Manager as a model. It was added into the QDP produced by the follow-on IV&V activity.

### 15.4.1 API Model

File: `barrier-mgr-model.pml`

Modelling waiting is much easier than modelling queueing. All that is required is an array of booleans (`waiters`), indexed by process id:

```
typedef Barrier {
  byte b_name; // barrier name
  bool isAutomatic; // true for automatic, false for manual barrier
  int maxWaiters; // maximum count of waiters for automatic barrier
  int waiterCount; // current amount of tasks waiting on the barrier
  bool waiters[TASK_MAX]; // waiters on the barrier
  bool isInitialised; // indicated whenever this barrier was created
}
```

The name `satisfied` is currently used here for an inline that checks when a barrier can be released. Other helper inlines include `waitAtBarrier()` and `barrierRelease()`.

### 15.4.2 Behaviour Patterns

File: `barrier-mgr-model.pml`

The overall architecture in terms of Promela processes has processes `init`, `System`, `Clock`, `Runner`, and two workers: `Worker1` and `Worker2`. The runner and workers each may perform one or more API calls, in the following order: create, ident, wait, release, delete. Scenarios mix and match which task does what.

There are three top-level scenarios:

```
mtype = {ManAcqRel,AutoAcq,AutoToutDel};
```

In scenario `ManAcqRel`, the runner will do create, release and delete, with sub-scenarios to check error cases as well as good behaviour, for manual barriers. Good behaviour involves one or more workers doing a wait. The `AutoAcq` and `AutoToutDel` scenarios look at good and bad uses of automatic barriers.

### 15.4.3 Annotations

File: `barrier-mgr-model.pml`

Similiar to those used in the Event Manager.

### 15.4.4 Refinement

File: `barrier-mgr-model-rfn.yml`

Similiar to those used in the Event Manager.

## 15.5 Testing the Message Manager

Documentation: Message Manager section in the RTEMS Classic API Guide.

Model Directory: `formal/promela/models/messages`.

Model Name: `msg-mgr-model`.

The Message Manager provides objects that act as message queues. Tasks can interact with these by enqueuing and/or dequeuing message objects.

There are 11 directives in total, but only the following were modelled:

- `rtems_message_queue_create()`
- `rtems_message_queue_send()`
- `rtems_message_queue_receive()`

The manager supports two queuing protocols, FIFO and priority-based. Only the FIFO queueing was modelled.

This model was developed by a Masters student [Lyn22], using the Event Manager as a model. It was added into the QDP produced by the follow-on IV&V activity.

Below we focus on aspects of this model that differ from the Event Manager.

### 15.5.1 API Model

File: `msg-mgr-model.pml`

A key feature of this manager is that not only are messages in a queue, but so are the tasks waiting for those messages. Both task and message queues are modelled as circular buffers, with inline definitions of enqueuing and dequeuing operations.

While the Message Manager allows many queues to be created, the model only uses one.

### 15.5.2 Behaviour Patterns

File: `msg-mgr-model.pml`

The overall architecture in terms of Promela processes has processes `init`, `System`, `Clock`, `Sender`, and two receivers: `Receiver1` and `Receiver2`. The `Sender` is the test runner, which performs the queue creation, releases the start semaphore and then performs a send operation if needed. The receivers are worker processes.

There are four top level scenarios:

```
1 mtype = {Send,Receive,SndRcv, RcvSnd};
```

Scenarios `Send` and `Receive` are used for testing erroneous calls. The `SndRcv` scenario fills up queues before the receivers run, while the `RcvSnd` has the receivers waiting before messages are sent.

### 15.5.3 Annotations

File: `msg-mgr-model.pml`

Similiar to those used in the Event Manager.

### 15.5.4 Refinement

File: `msg-mgr-model-rfn.yml`

Similiar to those used in the Event Manager.

## 15.6 Current State of Play

The models developed here are the result of an ad-hoc incremental development process and have a lot of overlapping material.

### 15.6.1 Model State

The models were developed by first focusing on simple behavior such as error handling, and then adding in simpler behaviors, until sufficient coverage was acheived. The basic philosophy at the time was not to fix anything not broken.

This has led to the models being somewhat over-engineered, particularly when it comes to scenario generation. There is some conditional looping behaviour, implemented using labels and `goto`, that should really be linearised, using conditionals to skip parts. It is harder than it should be to understand what each scenario does.

Also the API call models have perhaps a bit too much code devoted to system behaviour.

### 15.6.2 Model Refactoring

There is a case to be made to perform some model refactoring. Some of this would address the model state issues above. Other refactoring would extract the common model material out, to be put into files that could be included. This includes the binary semaphore models, and the parts modelling preemption and waiting while blocked.

### 15.6.3 Test Code Refactoring

During the qualification activity, a new file `tx-support.c` was added to the RTEMS validation testsuite codebase. This gathers C test support functions used by most of the tests. The contents of the `tr-<modelname>.h` and `tr-<modelname>.c` files in particular should be brought in line with `tx-support.c`.

Suitable Promela models should also be produced of relevant functions in `tx-support.c`.

CHAPTER

**SIXTEEN**

# GLOSSARY

**API**
This term is an acronym for Application Programming Interface.

**assembler language**
The assembler language is a programming language which can be translated very easily into machine code and data. For this project assembler languages are restricted to languages accepted by the *GNU* assembler program for the target architectures.

**C language**
The C language for this project is defined in terms of *C11*.

**C11**
The standard ISO/IEC 9899:2011.

**CCB**
This term is an acronym for Change Control Board.

**Doorstop**
Doorstop is a requirements management tool.

**EARS**
This term is an acronym for Easy Approach to Requirements Syntax.

**ELF**
This term is an acronym for Executable and Linkable Format.

**formal model**
A model of a computing component (hardware or software) that has a mathematically based *semantics*.

**GCC**
This term is an acronym for GNU Compiler Collection.

**GNAT**
*GNAT* is the *GNU* compiler for Ada, integrated into the *GCC*.

**GNU**
This term is an acronym for GNU's Not Unix.

**interrupt service**
An *interrupt service* consists of an *Interrupt Service Routine* which is called with a user provided argument upon reception of an interrupt service request. The routine is invoked in interrupt context. Interrupt service requests may have a priority and an affinity to a set of processors. An *interrupt service* is a *software component*.

**Interrupt Service Routine**
An ISR is invoked by the CPU to process a pending interrupt.

**ISVV**
This term is an acronym for Independent Software Verification and Validation.

**Linear Temporal Logic**
This is a logic that states properties about (possibly infinite) sequences of states.

**LTL**
This term is an acronym for *Linear Temporal Logic*.

**refinement**
A *refinement* is a relationship between a specification and its implementation as code.

**reification**
Another term used to denote *refinement*.

**ReqIF**
This term is an acronym for Requirements Interchange Format.

**RTEMS**
This term is an acronym for Real-Time Executive for Multiprocessor Systems.

**scenario**
In the context of formal verification, in a setting that involves many concurrent tasks that interleave in arbitrary ways, a scenario describes a single specific possible interleaving. One interpretation of the behaviour of a concurrent system is the set of all its scenarios.

**semantics**
This term refers to the meaning of text or utterances in some language. In a software engineering context these will be programming, modelling or specification languages.

**software component**
This term is defined by ECSS-E-ST-40C 3.2.28 as a "part of a software system". For this project a *software component* shall be any of the following items and nothing else:

- *software unit*
- explicitly defined *ELF* symbol in a *source code* file
- *assembler language* data in a source code file
- *C language* object with static storage duration
- C language object with thread-local storage duration
- *thread*
- *interrupt service*
- collection of *software components* (this is a software architecture element)

Please note that explicitly defined ELF symbols and assembler language data are considered a software component only if they are defined in a *source code* file. For example, this rules out symbols and data generated as side-effects by the toolchain (compiler, assembler, linker) such as jump tables, linker trampolines, exception frame information, etc.

**software product**
The *software product* is the *RTEMS* real-time operating system.

**software unit**
This term is defined by ECSS-E-ST-40C 3.2.24 as a "separately compilable piece of source code". For this project a *software unit* shall be any of the following items and nothing else:

- *assembler language* function in a *source code* file
- *C language* function (external and internal linkage)

A *software unit* is a *software component*.

**source code**
This project uses the *source code* definition of the Linux Information Project: "Source code (also referred to as source or code) is the version of software as it is originally written (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters)."

**target**
The system on which the application will ultimately execute.

**task**
This project uses the thread definition of Wikipedia: "a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system."

It consists normally of a set of registers and a stack. The scheduler assigns processors to a subset of the ready tasks. The terms task and *thread* are synonym in RTEMS. The term task is used throughout the Classic API, however, internally in the operating system implementation and the POSIX API the term thread is used.

A *task* is a *software component*.

**thread**
This term has the same meaning as *task*.

**YAML**
This term is an acronym for YAML Ain't Markup Language.

CHAPTER

**SEVENTEEN**

# REFERENCES

# BIBLIOGRAPHY


[Bra97] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. BCP 14, RFC Editor, March 1997. http://www.rfc-editor.org/rfc/rfc2119.txt. URL: http://www.rfc-editor.org/rfc/rfc2119.txt.

[BA14] Jace Browning and Robert Adams. Doorstop: Text-Based Requirements Management Using Version Control. *Journal of Software Engineering and Applications*, 7:187–194, 2014. URL: http://www.scirp.org/pdf/JSEA_2014032713545074.pdf.

[BH21] Andrew Butterfield and Mike Hinchey. *FV1-200: Formal Verification Plan*. Lero – the Irish Software Research Centre, 2021.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975. URL: https://doi.org/10.1145/360933.360975, doi:10.1145/360933.360975.

[ECS09] ECSS. *ECSS-E-ST-10-06C - Technical requirements specification*. European Cooperation for Space Standardization, 2009. URL: https://ecss.nl/standard/ecss-e-st-10-06c-technical-requirements-specification/.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul J. Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, 2009. URL: https://doi.org/10.1145/1459352.1459354, doi:10.1145/1459352.1459354.

[Jaskuc22] Jerzy Jaśkuć. SPIN/Promela-Based Test Generation Framework for RTEMS Barrier Manager. Master's thesis, School of Computer Science and Statistics, Trinity College, Dublin 2, Ireland, April 2022.

[Jen21] Robert Jennings. Formal Verification of a Real-Time Multithreaded Operating System. Master's thesis, School of Computer Science and Statistics, Trinity College, Dublin 2, Ireland, August 2021.

[Lyn22] Eoin Lynch. Using Promela/SPIN to do Test Generation for RTEMS-SMP. Master's thesis, School of Engineering, Trinity College, Dublin 2, Ireland, April 2022.

[MW10] Alistair Mavin and Philip Wilkinson. Big Ears (The Return of Easy Approach to Requirements Engineering). In *18th Requirements Engineering Conference*, 277–282. 11 2010. URL: https://www.researchgate.net/profile/Alistair_Mavin/publication/224195362_Big_Ears_The_Return_of_Easy_Approach_to_Requirements_Engineering/links/568ce39808ae197e426a075e/

Big-Ears-The-Return-of-Easy-Approach-to-Requirements-Engineering.pdf, doi:10.1109/RE.2010.39.

[MWGU16] Alistair Mavin, Philip Wilkinson, Sarah Gregory, and Eero Uusitalo. Listens Learned (8 Lessons Learned Applying EARS). In *24th International Requirements Engineering Conference*. September 2016. URL: https://www.researchgate.net/profile/Alistair_Mavin/publication/308970788_Listens_Learned_8_Lessons_Learned_Applying_EARS/links/5ab0d42caca2721710fe5017/Listens-Learned-8-Lessons-Learned-Applying-EARS.pdf, doi:10.1109/RE.2016.38.

[MWHN09] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (EARS). In *17th Requirements Engineering Conference*, 317–322. 10 2009. URL: https://www.researchgate.net/profile/Alistair_Mavin/publication/224079416_Easy_approach_to_requirements_syntax_EARS/links/568ce3bf08aeb488ea311990/Easy-approach-to-requirements-syntax-EARS.pdf, doi:10.1109/RE.2009.9.

[Mot88] Motorola. *Real Time Executive Interface Definition*. Motorola Inc., Microcomputer Division and Software Components Group, Inc., January 1988. DRAFT 2.1. URL: https://ftp.rtems.org/pub/rtems/publications/RTEID-ORKID/RTEID-2.1/RTEID-2_1.pdf.

[VIT90] VITA. *Open Real-Time Kernel Interface Definition*. VITA, the VMEbus International Trade Association, August 1990. Draft 2.1. URL: https://ftp.rtems.org/pub/rtems/publications/RTEID-ORKID/ORKID-2.1/ORKID-2_1.pdf.

[WB13] Karl Wiegers and Joy Beatty. *Software Requirements*. Microsoft Press, 3 edition, 2013. ISBN 0735679665, 9780735679665.

# INDEX