RTEMS Software Engineering

Release 5.c5749d0-modified (1st November 2019)

CONTENTS

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

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://devel.rtems.org/wiki/Developer/Bug_Reporting
Git Repositories	https://git.rtems.org
Developers	https://devel.rtems.org

CONTENTS

PREFACE

The Real Time Executive for Multiprocessor Systems (RTEMS) operating systems is a layered system with each of the public APIs implemented in terms of a common foundation layer called the SuperCore. RTEMS provides full capabilities for management of tasks, interrupts time, and multiple processors in addition to those features typical of generic operating systems. RTEMS has been implemented in both the Ada and C programming languages.

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

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 too. 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.

RTEMS STAKEHOLDERS

RTEMS is a community based 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.

3.1 Qualification - 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 stakeholder to invest in an ongoing basis to ensure the any investment they make is maintained and viable in an ongoing basis. The RTEMS core developers view steady support of the qualification effort as necessary to continue to lower the overall costs of qualification RTEMS.

SOFTWARE DEVELOPMENT MANAGEMENT

4.1 Software Development (Git Users)

4.1.1 Browse the Git Repository Online

You can browse all available repositories online by accessing https://git.rtems.org/.

4.1.2 Using the Git Repository

The following examples demonstrate how to use the RTEMS' Git repos. These examples are provided for the main rtems module, but they are also valid for the other modules.

First, we need to obtain our own local copy of the RTEMS Git repository:

```
git clone git://git.rtems.org/rtems.git rtems
```

This command will create a folder named rtems in the current directory. This folder will contain a full-featured RTEMS' Git repository and the current HEAD revision checked out. Since all the history is available we can check out any release of RTEMS. Major RTEMS releases are available as separate branches in the repo.

To see all available remote branches issue the following command:

```
git branch -r
```

We can check out one of those remote branches (e.g. rtems-4.10 branch) using the command:

```
git checkout -b rtems410 origin/4.10
```

This will create a local branch named "rtems410", containing the rtems-4.10 release, that will track the remote branch "rtems-4-10-branch" in origin (git://git.rtems.org/rtems.git). The `git branch` command prints a list of the current local branches, indicating the one currently checked out.

If you want to switch between local branches:

```
git checkout <branch-name>
```

With time your local repository will diverge from the main RTEMS repository. To keep your local copy up to date you need to issue:

```
git pull origin
```

This command will update all your local branches with any new code revisions available on the central repository.

4.1.3 Making Changes

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 master branch track the upstream RTEMS' master branch without interacting with any of the changes you are working on. A completed change

is emailed to the developer's list for review and this can take time. While this is happening the upstream's master 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.

First, you need to clone the repository:

```
1 git clone git://git.rtems.org/rtems.git rtems
```

Or if you already cloned it before, then you might want to update to the latest version before making your changes:

```
1 cd rtems
2 git pull
```

Create a local branch to make your changes in, in this example, the change is `faster-context-switch`:

```
1 git checkout -b faster-context-switch
```

Next, make your changes to files. If you add, delete ormove/rename files you need to inform Git

```
1 git add /some/new/file
2 git rm /some/old/file
3 git mv /some/old/file /some/new/file
```

When you're satisfied with the changes you made, commit them (locally)

```
1 git commit -a
```

The `-a` flag commits all the changes that were made, but you can also control which changes to commit by individually adding files as you modify them by using. You can also specify other options to commit, such as a message with the `-m` flag.

```
1 git add /some/changed/files
2 git commit
```

Create a patch from your branch, in this case, we have two commits we want to send for review:

```
1   git format-patch -2
2
3 There are new changes pushed to the RTEMS' master branch and our local branch
4 needs to be updated:
```

```
1 git checkout master
2 git pull
3 git checkout faster-context-switch
4 git rebase master
```

4.1.4 Working with Branches

Branches facilitate trying out new code and creating patches.

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:

```
1 git branch -r
```

Then check out the desired remote branch, for example:

```
1 git checkout -b rtems410 origin/4.10
```

Or if you have previously checked out the remote branch then you should see it in your local branches:

```
1 git branch
```

You can change to an existing local branch easily:

```
1 git checkout rtems410
```

You can also create a new branch and switch to it:

```
1 git branch temporary
2 git checkout temporary
```

Or more concisely:

```
1 git checkout -b temporary
```

If you forget which branch you are on

```
1 git branch
```

shows you by placing a * next to the current one.

When a branch is no longer useful you can delete it.

```
1 git checkout master
2 git branch -d temporary
```

If you have unmerged changes in the old branch Git complains and you need to use -D instead of -d.

4.1.5 Viewing Changes

To view all changes since the last commit:

```
1 git diff HEAD
```

To view all changes between the current branch and another branch, say master:

```
1 git diff master..HEAD
```

To view descriptions of committed changes:

```
1 git log
```

Or view the changeset for some file (or directory):

```
1  git log /some/file
```

To view the changesets made between two branches:

```
1  git log master..HEAD
```

Or for a more brief description use shortlog:

```
1  git shortlog master..HEAD
```

4.1.6 Reverting Changes

To remove all (uncommitted) changes on a branch

```
1  git checkout -f
```

Or to selectively revert (uncommited) files, for example if you accidentally deleted ./some/file

```
1  git checkout -- ./some/file
```

or

```
1  git checkout HEAD ./some/file
```

To remove commits there are two useful options, reset and revert. `git reset` should only be used on local branches that no one else is accessing remotely. `git revert` is cleaner and is the right way to revert changes that have already been pushed/pulled remotely.

4.1.7 git reset

`git reset` is a powerful and tricky command that should only be used on local (un-pushed) branches): A good description of what it enables to do can be found here. The following are a few useful examples. Note that adding a ~ after HEAD refers to the most recent commit, and you can add a number after the ~ to refer to commits even further back; HEAD by itself refers to the current working directory (changes since the last commit).

```
1  git reset HEAD~
```

Will undo the last commit and unstage those changes. Your working directory will remain the same, therefore a `git status` will yield any changes you made plus the changes made in your last commit. This can be used to fix the last commit. You will need to add the files again.

```
1  git reset --soft HEAD~
```

Will just undo the last commit. The changes from the last commit will still be staged (just as if you finished git adding them). This can be used to amend the last commit (e.g. You forgot to add a file to the last commit).

```
1  git reset --hard HEAD~
```

Will revert everything, including the working directory, to the previous commit. This is dangerous and can lead to you losing all your changes; the `--hard` flag ignores errors.

```
1 git reset HEAD
```

Will unstage any change. This is used to revert a wrong `git add`. (e.g. You added a file that shouldn't be there, but you haven't 'committed')

Will revert your working directory to a HEAD state. You will lose any change you made to files after the last commit. This is used when you just want to destroy all changes you made since the last commit.

4.1.8 git revert

`git revert` does the same as reset but creates a new commit with the reverted changes instead of modifying the local repository directly.

```
1 git revert HEAD
```

This will create a new commit which undoes the change in HEAD. You will be given a chance to edit the commit message for the new commit.

4.1.9 Merging Changes

Suppose you commit changes in two different branches, branch1 and branch2, and want to create a new branch containing both sets of changes:

```
1 git checkout -b merged
2 git merge branch1
3 git merge branch2
```

Or you might want to bring the changes in one branch into the other:

```
1 git checkout branch1
2 git merge branch2
```

And now that branch2 is merged you might get rid of it:

```
1 git branch -d branch2
```

If you have done work on a branch, say branch1, and have gone out-of-sync with the remote repository, you can pull the changes from the remote repo and then merge them into your branch:

```
1 git checkout master
2 git pull
3 git checkout branch1
4 git merge master
```

If all goes well the new commits you pulled into your master branch will be merged into your branch1, which will now be up-to-date. However, if branch1 has not been pushed remotely then rebasing might be a good alternative to merging because the merge generates a commit.

4.1.10 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.

For example

```
1 git checkout branch1
2 git rebase master
```

or more concisely

```
1 git rebase master branch1
```

will bring the changes of master into branch1, and then you can fast-forward master to include branch1 quite easily

```
1 git checkout master
2 git merge branch1
```

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.

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.

4.1.11 Accessing a developer's repository

RTEMS developers with Git commit access have personal repositories on https://git.rtems.org/ that can be cloned to view cutting-edge development work shared there.

4.1.12 Creating a Patch

Before submitting a patch read about Contributing to RTEMS and the Commit Message formatting we require.

The recommended way to create a patch is to branch the Git repository master and use one commit for each logical change. Then you can use `git format-patch` to turn your commits into patches and easily submit them.

```
1 git format-patch master
```

Creates a separate patch for each commit that has been made between the master branch and the current branch and writes them in the current directory. Use the `-o` flag to redirect the files to a different directory.

If you are re-submitting a patch that has previously been reviewed, you should specify a version number for your patch, for example, use

```
git format-patch -v2 ...
```

to indicate the second version of a patch, `-v3` for a third, and so forth.

Patches created using `git format-patch` are formatted so they can be emailed and rely on having Git configured with your name and email address, for example

```
git config --global user.name "Your Name"
git config --global user.email name@domain.com
```

Please use a real name, we do not allow pseudonyms or anonymous contributions.

4.1.13 Submitting a Patch

Using `git send-email` you can easily contribute your patches. You will need to install `git send-email` first:

```
sudo yum install git-email
```

or

```
sudo dnf install git-email
```

or

```
sudo apt install git-email
```

Then you will need to configure an SMTP server. You could install one on your localhost, or you can connect to a mail server such as Gmail.

4.1.14 Configuring git send-email to use Gmail

Configure Git to use Gmail:

```
git config --global sendemail.smtpserver smtp.gmail.com
git config --global sendemail.smtpserverport 587
git config --global sendemail.smtpencryption tls
git config --global sendemail.smtpuser your_email@gmail.com
```

It will ask for your password each time you use `git send-email`. Optionally you can also put it in your `git config`:

```
git config --global sendemail.smtppass your_password
```

4.1.15 Sending Email

To send your patches just

```
1 git send-email /path/to/patch --to devel@rtems.org
```

To send multiple related patches (if you have more than one commit in your branch) specify a path to a directory containing all of the patches created by `git format-patch`. `git send-email` has some useful options such as:

- `--annotate` to show/edit your patch

- `--cover-letter` to prepend a summary

- `--cc=<address>` to cc someone

You can configure the to address:

```
1 git config --global sendemail.to devel@rtems.org
```

So all you need is:

```
1 git send-email /path/to/patch
```

4.1.16 Troubleshooting

Some restrictive corporate firewalls block access through the Git protocol (git://). If you are unable to reach the server git://git.rtems.org/ you can try accessing through http. To clone the rtems repository using the http protocol use the following command:

```
1 git clone http://git.rtems.org/rtems/ rtems
```

This access through http is slower (way slower!) than through the git protocol, therefore, the Git protocol is preferred.

4.1.17 Manage Your Code

You may prefer to keep your application and development work in a Git repository for all the good reasons that come with version control. For public repositories, you may like to try GitHub or BitBucket. RTEMS maintains mirrors on GitHub which can make synchronizing with upstream changes relatively simple. If you need to keep your work private, you can use one of those services with private repositories or manage your own server. The details of setting up a server are outside the scope of this document, but if you have a server with SSH access you should be able to find instructions on how to set up Git access. Once you have git configured on the server, adding repositories is a snap.

4.1.18 Private Servers

In the following, replace @USER@ with your username on your server, @REPO@ with the name of your repository, and @SERVER@ with your server's name or address.

To push a mirror to your private server, first create a bare repository on your server.

```
1 cd /home/@USER@
2 mkdir git
3 mkdir git/@REPO@.git
```

```
4 cd git/@REPO@.git
5 git --bare init
```

Now from your client machine (e.g. your work laptop/desktop), push a git, perhaps one you cloned from elsewhere, or one that you made locally with `git init`, by adding a remote and pushing:

```
1 git remote add @SERVER@ ssh://@SERVER@/home/@USER@/git/@REPO@.git
2 git push @SERVER@ master
```

You can replace the @SERVER@ with another name for your remote if you like. And now you can push other branches that you might have created. Now you can push and pull between your client and your server. Use SSH keys to authenticate with your server if you want to save on password typing; remember to put a passphrase on your SSH key if there is a risk the private key file might get compromised.

The following is an example scenario that might be useful for RTEMS users that uses a slightly different approach than the one just outlined:

```
1  ssh @SERVER@
2  mkdir git
3  git clone --mirror git://git.rtems.org/rtems.git
4  ## Add your ssh key to ~/.ssh/authorized_keys
5  exit
6  git clone ssh://@SERVER@/home/@USER@/git/rtems.git
7  cd rtems
8  git remote add upstream git://git.rtems.org/rtems.git
9  git fetch upstream
10 git pull upstream master
11 git push
12 ## If you want to track RTEMS on your personal master branch,
13 ## you should only push changes to origin/master that you pull
14 ## from upstream. The basic workflow should look something like:
15 git checkout master
16 git pull upstream master
17 git push
18 git checkout -b anewbranch
19 ## Repeat: do work, git commit -a
20 git push origin anewbranch
21
22 ## delete a remote branch
23 git push origin :anewbranch
24 ## delete a local branch
25 git branch -d anewbranch
```

4.1.19 Learn more about Git

Links to the sites with good Git information:

- http://gitready.com/ - An excellent resource from beginner to very advanced.

- http://progit.org/book/ - Covers Git basics and some advanced features. Includes some useful workflow examples.

- https://lab.github.com/ - Learn to use Git and GitHub while doing a series of projects.

- https://git-scm.com/docs - The official Git reference.

4.2 Software Development (Git Writers)

4.2.1 SSH Access

Currently all committer's should have an ssh account on the main git server, dispatch.rtems.org. If you have been granted commit access and do have an account on dispatch.rtems.org one should be requested on the devel@ list. SSH access for git uses key logins instead of passwords. The key should be at least 1024 bits in length.

The public repositories can by cloned with

```
1 git clone ssh://user@dispatch.rtems.org/data/git/rtems.git
```

Or replace *rtems.git* with another repo to clone another one.

4.2.2 Personal Repository

Personal repositories keep the clutter away from the master repository. A user with a personal repository can make commits, create and delete branches, plus more without interfering with the master repository. Commits to the master repository generate email to the vc@ list and development type commits by a developer would only add noise and lessen the effectiveness of the commit list

A committer should maintain a personal clone of the RTEMS repository through which all changes merged into the RTEMS head are sent. The personal repository is also a good place for committers to push branches that contain works in progress. The following instructions show how to setup a personal repositor that by default causes commits to go to your private local repository and pushes to go to your publicly visible personal repository. The RTEMS head is configured as a remote repository named 'upstream' to which you can push changes that have been approved for merging into RTEMS.

Branches aren't automatically pushed until you tell git to do the initial push after which the branch is pushed automatically. In order to keep code private just put it on a branch in your local clone and do not push the branch.

4.2.3 Create a personal repository

Set up the server side repository. In the following substitute user with your username.

```
1 # ssh git.rtems.org
2 [user@git ~]$ ln -s /data/git/user git
3 [user@git ~]$ ls -l
4 lrwxrwxrwx 1 user rtems 16 Feb  1 11:52 git -> /data/git/user
5 [user@git ~]$ cd git
6 [user@git git]$ git clone --mirror /data/git/rtems.git
```

Provide a description for the repository, for example "Clone of master repository."

```
1 [user@git git]$ echo "Clone of master repository." > rtems.git/description
2 [user@git git]$ logout
```

Clone the repository on your local machine

```
1  # git clone ssh://user@dispatch.rtems.org/home/user/git/rtems.git
2  # cd rtems
```

Add the RTEMS repository as a remote repository and get the remote tags and branches

```
1  # git remote add upstream ssh://user@dispatch.rtems.org/data/git/rtems.git
2  # git fetch upstream
```

After a little while you should be able to see your personal repo at https://git.rtems.org/@USER@/rtems.git/ and you can create other repositories in your git directory that will propagate to https://git.rtems.org/@USER@/ if you need. For example, *joel*'s personal repos appear at https://git.rtems.org/joel/.
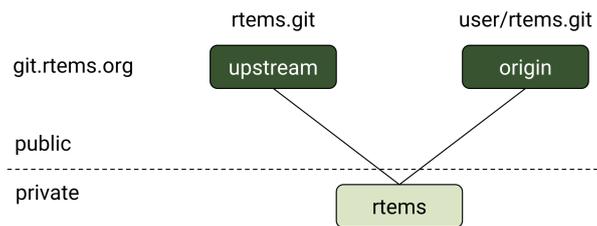
4.2.3.1 Check your setup

```
1  git remote show origin
```

Should print something similar to

```
1  * remote origin
2    Fetch URL: ssh://user@dispatch.rtems.org/home/user/git/rtems.git
3    Push  URL: ssh://user@dispatch.rtems.org/home/user/git/rtems.git
4    HEAD branch: master
5    Remote branches:
6      4.10   tracked
7      4.8    tracked
8      4.9    tracked
9      master tracked
10   Local branch configured for 'git pull':
11     master merges with remote master
12   Local ref configured for 'git push':
13     master pushes to master (up to date)
```

4.2.3.2 Push commits to personal repo master from local master

```
1  # git push
```

4.2.3.3 Push a branch onto personal repo

```
1  # git push origin branchname
```

4.2.3.4 Update from upstream master (RTEMS head)

When you have committed changes on a branch that is private (hasn't been pushed to your personal repo) then you can use rebase to obtain a linear history and avoid merge commit messages.

```
# git checkout new_features
# git pull --rebase upstream master
```

If you cannot do a fast-forward merge then you could use the `--no-commit` flag to prevent merge from issuing an automatic merge commit message.

When you have committed changes on a branch that is public/shared with another developer you should not rebase that branch.

4.2.4 GIT Push Configuration

People with write access to the main repository should make sure that they push the right branch with the git push command. The above setup ensures that git push will not touch the main repository, which is identified as upstream, unless you specify the upstream (by `git push upstream master`).

Lets suppose we have a test branch intended for integration into the master branch of the main repository.

```
# git branch
  master
* test
```

There are two options for pushing with the branch. First,

```
# git push origin test
```

Will push the test branch to the personal repository. To delete the remote branch

```
# git push origin :test
```

You'll still need to delete your local branch if you are done with it.

If you are going to work exclusively with one branch for a while, you might want to configure git to automatically push that branch when you use git push. By default git push will use the local master branch, but you can use the *test* branch as the source of your changes:

```
# git config remote.origin.push test:master
```

Now git push will merge into your master branch on your personal repository. You can also setup a remote branch:

```
# git config remote.origin.push test:test
```

You can see what branch is configured for pushing with

```
# git remote show origin
```

And reset to the default

```
1  # git config remote.origin.push master
```

4.2.5 Pull a Developer's Repo

The procedures for creating personal repositories ensure that every developer can post branches that anyone else can review. To pull a developer's personal repository into your local RTEMS git clone, just add a new remote repo:

```
1  # git remote add devname git://dispatch.rtems.org/devname/rtems.git
2  # git fetch devname
3  # git remote show devname
4  # git branch -a
```

Replace devname with the developer's user name on git, which you can see by accessing https://git.rtems.org. Now you can switch to the branches for this developer.

Use a tracking branch if the developer's branch is changing:

```
1  # git branch --track new_feature devname/new_feature
```

4.2.6 Committing

4.2.6.1 Ticket Updates

Our trac instance supports updating a related ticket with the commit message.

Any references to a ticket for example #1234 will insert the message into he ticket as an 'update'. No command is required.

Closing a ticket can be done by prefixing the ticket number with any of the following commands:

`close`, `closed`, `closes`, `fix`, `fixed`, or `fixes`

For example:

`closes #1234`

`This is a random update it closes #1234 and updates #5678`

4.2.6.2 Commands

When merging someone's work, whether your own or otherwise, we have some suggested procedures to follow.

- Never work in the master branch. Checkout a new branch and apply patches/commits to it.

- Before pushing upstream: - Update master by fetching from the server - Rebase the working branch against the updated master - Push the working branch to the server master

The basic workflow looks like

```
1  # git checkout -b somebranch upstream/master
2  # patch .. git add/rm/etc
3  # git commit ...
4  # git pull --rebase upstream master
5  # git push upstream somebranch:master
```

If someone pushed since you updated the server rejects your push until you are up to date.

For example a workflow where you will commit a series of patches from `../patches/am/` directory:

```
1  # git checkout -b am
2  # git am ../patches/am*
3  # git pull --rebase upstream master
4  # git push upstream am:master
5  # git checkout master
6  # git pull upstream master
7  # git log
8  # git branch -d am
9  # git push
```

The git log stage will show your newly pushed patches if everything worked properly, and you can delete the am branch created. The git push at the end will push the changes up to your personal repository.

Another way to do this which pushes directly to the upstream is shown here in an example which simply (and quickly) applies a patch to the branch:

```
1  git checkout -b rtems4.10 --track remotes/upstream/4.10
2  cat /tmp/sp.diff | patch
3  vi sparc.t
4  git add sparc.t
5  git commit -m "sparc.t: Correct for V8/V9"
6  git push upstream rtems4.10:4.10
7  git checkout master
8  git log
9  git branch -d rtems4.10
```

4.2.7 Pushing Multiple Commits

A push with more than one commit results in Trac missing them. Please use the following script to push a single commit at a time:

```
1  #! /bin/sh
2  commits=$(git log --format='%h' origin/master..HEAD | tail -r)
3  for c in $commits
4  do
5    cmd=$(echo $c | sed 's%\(.*\)%git push origin \1:master%')
6    echo $cmd
7  $cmd
8  done
```

4.2.8 Ooops!

So you pushed something upstream and broke the repository. First things first: stop what you're doing and notify devel@... so that (1) you can get help and (2) no one pulls from the broken repo. For an extended outage also notify users@.... Now, breathe easy and let's figure out what happened. One thing that might work is to just undo the push. To get an idea of what you did, run `git reflog`, which might be useful for getting assistance in undoing whatever badness was done.

4.3 Coding Standards

TBD - Write introduction, re-order, identify missing content

4.3.1 Coding Conventions

The style of RTEMS is generally consistent in the core areas. This page attempts to capture generally accepted practices. When in doubt, consult the code around you or look in cpukit/rtems. See the sister page Doxygen Recommendations. for examples that illustrate style rules and Doxygen usage.

4.3.1.1 Source Documentation

- Use Doxygen according to our Doxygen Recommendations..

- Start each file with a brief description followed by a license. See Boilerplate File Header..

- 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.

- Use XXX or FIXME to indicate an error/bug/broken code.

4.3.1.2 Licenses

- The RTEMS License. is the typical and preferred license. * 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.

  – GPL licensed code is NOT acceptable, neither is LGPL. See this blog post explanation. for more information.

  – Advertising obligations are NOT acceptable, but restrictions are permissible.

4.3.1.3 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 score/basedefs.h for abstracting compiler-specific features.

- Use NULL for the null pointer, and prefer to use explicit checks against NULL, e.g.,

```
1  if ( ptr != NULL )
```

instead of

```
1  if ( !ptr )
```

- **Use explicit checks for bits in variables.**

  - **Example 1: Use**

    ```
    1  if ( XBITS == (var & XBITS) )
    ```

    to check for a set of defined bits.

  - **Example 2: Use**

    ```
    1  if ( (var & X_FLAGS) != 0) )
    ```

    instead of

    ```
    1  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.

4.3.1.4 Formatting

- Use spaces instead of tabs.

- Use two spaces for indentation, four spaces for hanging indentation.

- Adhere to a limit of 80 characters per line..

- 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 parens.

- 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 whitespace at the end of a line.

4.3.1.5 Readability

- Understand and follow the naming rules..

- 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 feature-conditioned compilation..

- 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.

4.3.1.6 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..

4.3.1.7 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.

4.3.1.8 Maintainability

- Minimize modifications to third-party 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.

4.3.1.9 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).

- Functions used only through function pointers should be declared 'static inline' (RTEMS_INLINE_ROUTINE)

- Prefer to ++preincrement instead of postincrement++.

- Avoid using floating point except where absolutely necessary.

4.3.1.10 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.

4.3.1.11 Layering

- TBD: add something about the dependencies and header file layering.

- Understand the 'RTEMS Software Architecture <https://devel.rtems.org/wiki/TBR/ UserManual/RTEMS_Software_Architecture>'_.

4.3.1.12 Exceptions to the Rules

- Minimize reformatting existing code in RTEMS unless the file undergoes substantial non-style changes.

- Third-party code. should not be reformatted to fit RTEMS style. Exception: unmaintained third-party code adopted and maintained by RTEMS may be reformatted, subject to the above rules.

4.3.1.13 Tools

Some of the above can be assisted by tool support. Feel free to add more tools, configurations, etc here.

- Uncrustify. Configuration for RTEMS: rtems.uncrustify.

4.3.2 Eighty Character Line 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.

4.3.2.1 Breaking long lines

`if`, `while`, and `for` loops 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 (optional) 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. Here is an example
with a `for` loop.

```
for ( initialization = statement; a + really + long + statement + that + evaluates + to <␣
↪a + boolean; another + statement++ ) {
  z = a + really + long + 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 + long + statement + that + evaluates + to <
  a + boolean;
  another + statement++
) {
  z = a + really + long + statement + that + needs +
      two + lines + gets + indented + four + more +
      spaces + on + the + second + and + subsequent +
      lines + and + broken + up + at + operators;
}
```

Note that indentations should add 2 nesting levels (4 space characters, not tabs).

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

```
1  #define A_LONG_MACRO_NAME (AND + EXPANSION) /* Plus + a + really + long + comment */
```

can be replaced with

```
1  /* Plus + a + really + long + comment */
2  #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

```
1  #define A_LONG_MACRO_NAME (AND + EXCESSIVELY + LONG + EXPANSION + WITH + LOTS + OF + ␣
   ↪EXTRA + STUFF + DEFINED)
```

would become

```
1  #define A_LONG_MACRO_NAME ( \
2    AND + EXCESSIVELY + LONG + EXPANSION + WITH + LOTS + OF + EXTRA + STUFF + \
3    DEFINED \
4  )
```

Notice that each line is terminated by a backslash then the carriage return. 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

```
1  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

```
1  int this_is_a_function(
2    int arg1,
3    int arg2,
4    int arg3,
5    int arg4,
6    int arg5,
7    int arg6,
8    int arg7,
9    int arg8,
10   int arg9
11 );
```

Excessively long comments should be broken up at a word boundary or somewhere that makes sense, for example

```
1  /* Excessively long comments should be broken up at a word boundary or somewhere that␣
   ↪makes sense, for example */
```

would be

```
1  /* Excessively long comments should be broken up at a word boundary or
2   * somewhere that makes sense, for example */
```

Note that multiline comments have a single asterisk aligned with the asterisk in the opening /*. The closing */ should go at the end of the last line.

4.3.3 Deprectating Interfaces

TBD - Convert the following to Rest and insert into this file TBD - https://devel.rtems.org/wiki/Developer/Coding/Deprecating

4.3.4 Doxygen Guidelines

4.3.4.1 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.

```
1  /**
2   * @defgroup RTEMSScoreThread
3   *
4   * @ingrop RTEMSScore
5   *
6   * ...
7   */
```

4.3.4.2 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. The `@brief` description shall use Title Case. Use grammatically correct sentences for the detailed descriptions.

```
1  /**
2   * @defgroup RTEMSScoreThread
3   *
4   * @ingrop RTEMSScore
5   *
6   * @brief Thread Handler
7   *
8   * ...
9   *
10  * @{
11  */
12
13  ... declarations, defines ...
14
15  /** @} */
```

```
1  /**
2   * @addtogroup RTEMSScoreThread
3   *
4   * @{
```

```
5  */
6
7  ... declarations, defines ...
8
9  /** @} */
```

4.3.4.3 Files

Each source or header file shall have an `@file` block at the top of the file. The `@file` block should precede the license header separated by one blank line. This placement reduces the chance of merge conflicts in imported third-party code. The `@file` block shall be put into a group with `@ingroup GroupName`. The `@file` block should have an `@brief` description and a detailed description if it is considered helpful. Use `@brief @copybrief GroupName` as a default to copy the `@brief` description from the corresponding group and omit the detailed description.

```
1  /**
2   * @file
3   *
4   * @ingroup RTEMSScoreThread
5   *
6   * @brief @copybrief RTEMSScoreThread
7   */
```

```
1  /**
2   * @file
3   *
4   * @ingroup RTEMSScoreThread
5   *
6   * @brief Some helpful brief description.
7   *
8   * Some helpful detailed description.
9   */
```

4.3.4.4 Type Definitions

Each type defined in a header file shall be documented with an `@brief` description and an optional detailed description. Each type member shall be documented with an `@brief` description and an optional detailed description. Use grammatically correct sentences for the detailed descriptions.

```
1  /**
2   * @brief The information structure used to manage each API class of objects.
3   *
4   * If objects for the API class are configured, an instance of this structure
5   * is statically allocated and pre-initialized by OBJECTS_INFORMATION_DEFINE()
6   * through <rtems/confdefs.h>.  The RTEMS library contains a statically
7   * allocated and pre-initialized instance for each API class providing zero
8   * objects, see OBJECTS_INFORMATION_DEFINE_ZERO().
9   */
10 typedef struct {
11   /**
12    * @brief This is the maximum valid ID of this object API class.
```

```
13    *
14    * This member is statically initialized and provides also the object API,
15    * class and multiprocessing node information.
16    *
17    * It is used by _Objects_Get() to validate an object ID.
18    */
19   Objects_Id maximum_id;
20
21   ... more members ...
22 } Objects_Information;
```

4.3.4.5 Function Declarations

Each function declaration or function-like macros 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. 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 referenced object is modified by the function, or

- use @param[in, out], if the referenced object is read and modified by the function.

For other parameters (e.g. *const pointer* and *scalar* parameters) do not use the [in], [out] or [in, out] parameter specifiers. Each return value or return value range shall be documented with an @retval entry. Document the most common return value first. Use a placeholder name for value ranges, e.g. pointer in the _Workspace_Allocate() example below. In case the function returns only one value, then use @return, e.g. use @retval only if there are at least two return values or return value ranges. Use grammatically correct sentences for the parameter and return value descriptions.

```
1  /**
2   * @brief Sends a message to the message queue.
3   *
4   * This directive sends the message buffer to the message queue indicated by
5   * ID.  If one or more tasks is blocked waiting to receive a message from this
6   * message queue, then one will receive the message.  The task selected to
7   * receive the message is based on the task queue discipline algorithm in use
8   * by this particular message queue.  If no tasks are waiting, then the message
9   * buffer will be placed at the rear of the chain of pending messages for this
10  * message queue.
11  *
12  * @param id The message queue ID.
13  * @param buffer The message content buffer.
14  * @param size The size of the message.
15  *
16  * @retval RTEMS_SUCCESSFUL Successful operation.
17  * @retval RTEMS_INVALID_ID Invalid message queue ID.
18  * @retval RTEMS_INVALID_ADDRESS The message buffer pointer is @c NULL.
19  * @retval RTEMS_INVALID_SIZE The message size is larger than the maximum
20  *   message size of the message queue.
21  * @retval RTEMS_TOO_MANY The new message would exceed the message queue limit
22  *   for pending messages.
23  */
24 rtems_status_code rtems_message_queue_send(
```

```
25    rtems_id    id,
26    const void *buffer,
27    size_t      size
28 );
```

```
1  /**
2   * @brief Receives a message from the message queue
3   *
4   * This directive is invoked when the calling task wishes to receive a message
5   * from the message queue indicated by ID. The received message is to be placed
6   * in the buffer. If no messages are outstanding and the option set indicates
7   * that the task is willing to block, then the task will be blocked until a
8   * message arrives or until, optionally, timeout clock ticks have passed.
9   *
10  * @param id The message queue ID.
11  * @param[out] buffer The buffer for the message content.  The buffer must be
12  *   large enough to store maximum size messages of this message queue.
13  * @param[out] size The size of the message.
14  * @param option_set The option set, e.g. RTEMS_NO_WAIT or RTEMS_WAIT.
15  * @param timeout The number of ticks to wait if the RTEMS_WAIT is set.  Use
16  *   RTEMS_NO_TIMEOUT to wait indefinitely.
17  *
18  * @retval RTEMS_SUCCESSFUL Successful operation.
19  * @retval RTEMS_INVALID_ID Invalid message queue ID.
20  * @retval RTEMS_INVALID_ADDRESS The message buffer pointer or the message size
21  *   pointer is @c NULL.
22  * @retval RTEMS_TIMEOUT A timeout occurred and no message was received.
23  */
24 rtems_status_code rtems_message_queue_receive(
25    rtems_id        id,
26    void           *buffer,
27    size_t         *size,
28    rtems_option    option_set,
29    rtems_interval  timeout
30 );
```

```
1  /**
2   * @brief Allocates a memory block of the specified size from the workspace.
3   *
4   * @param size The size of the memory block.
5   *
6   * @retval pointer The pointer to the memory block.  The pointer is at least
7   *   aligned by CPU_HEAP_ALIGNMENT.
8   * @retval NULL No memory block with the requested size is available in the
9   *   workspace.
10  */
11 void *_Workspace_Allocate( size_t size );
```

```
1  /**
2   * @brief Rebalances the red-black tree after insertion of the node.
3   *
4   * @param[in, out] the_rbtree The red-black tree control.
5   * @param[in, out] the_node The most recently inserted node.
6   */
7  void _RBTree_Insert_color(
8     RBTree_Control *the_rbtree,
```

Chapter 4. Software Development Management

```
 9    RBTree_Node     *the_node
10 );
```

```
 1 /**
 2  * @brief Builds an object ID from its components.
 3  *
 4  * @param the_api The object API.
 5  * @param the_class The object API class.
 6  * @param node The object node.
 7  * @param index The object index.
 8  *
 9  * @return Returns the object ID constructed from the arguments.
10  */
11 #define _Objects_Build_id( the_api, the_class, node, index )
```

4.3.4.6 Header File Examples

The <rtems/score/thread.h> and <rtems/score/threadimpl.h> header files are a good exam-
ple of how header files should be documented.

4.3.5 Boilerplate File Header

Every file should include two comment header blocks, one for the Doxygen output and a copy-
right notice. This is a typical example:

```
 1 /**
 2  * @file
 3  *
 4  * @ingroup TheGroupForThisFile
 5  *
 6  * @brief Short "Table of Contents" Description of File Contents
 7  *
 8  * A short description of the purpose of this file.
 9  */
10
11 /*
12  * Copyright (c) 20XX Your Name Or Your Company.
13  *
14  * The license and distribution terms for this file may be
15  * found in the file LICENSE in this distribution or at
16  * https://www.rtems.org/license/LICENSE.
17  */
```

- Use exactly one blank line between the Doxygen header and copyright notice. Leave the
  first line of the copyright notice blank.

- Separate the Doxygen header and copyright notice so the copyright notice is not included
  in the Doxygen output.

- The copyright owner and specific license terms may vary.

4.3.6 Generating a Tools Patch

The RTEMS patches to the development tools are generated using a command like this

where the options are:

- -N and -P take care of adding and removing files (be careful not to

include junk files like file.mybackup)

- -r tells diff to recurse through subdirectories

- -c is a context diff (easy to read for humans)

- -u is a unified diff (easy for patch to apply)

Please look at the generated PATCHFILE and make sure it does not contain anything you did not intend to send to the maintainers. It is easy to accidentally leave a backup file in the modified source tree or have a spurious change that should not be in the PATCHFILE.

If you end up with the entire contents of a file in the patch and can't figure out why, you may have different CR/LF scheme in the two source files. The GNU open-source packages usually have UNIX style CR/LF. If you edit on a Windows platform, the line terminators may have been transformed by the editor into Windows style.

4.3.7 Naming Rules

4.3.7.1 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.
- Prefer to use underscores 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.

User-Facing APIs

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.

If the method is intended to be private, then make it static to a file or start the name with a leading _.

Classic API

- Public facing APIs start with *rtems_* followed by a word or phrase to indicate the Manager or functional category the method or data type belongs to.

- Non-public APIs should be static or begin with a leading _. The required form is the use of a leading underscore, functional area with leading capital letter, an underscore, and the method with a leading capital letter.

POSIX API

- Follow the rules of POSIX.

RTEMS Internal Interfaces

Super Core

The Super Core. is organized in an Object-Oriented fashion. Each score Handler 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 functions 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.

As shown in the following example:

```
1  typedef struct {
2      Other_module_Struct_type    Aggregate_member_name;
3      Other_module_Struct_type   *reference_member_name;
4      Other_module_Primitive_type primitive_member_name;
5    } The_module_Type_name;
```

BSP

- TODO.

4.4 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

4.5 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.

SOFTWARE TEST PLAN ASSURANCE AND PROCEDURES

5.1 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 master technical data does not bit rot.

Must use tools from:

TBD - Change URL to git.rtems.org and list support tools RTEMS Tools Project: https://github.com/RTEMS/rtems-tools

Scope, Procedures, Methodologies, Tools TBD - Write content

5.1.1 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.

5.1.1.1 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

5.1.2 RTEMS Tester

TBD - Convert the following to Rest and insert into this file TBD https://devel.rtems.org/wiki/Testing/Tester TBD - Above file is horribly out of date. Find new docs and refer to them

SOFTWARE TEST FRAMEWORK

6.1 The RTEMS Test Framework

The *RTEMS Test Framework* helps you to write test suites. It has the following features:

- Implemented in standard C11

- 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

6.1.1 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 the test action outcome is all right, then the test check passes, otherwise the 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.

6.1.2 Test Cases

You can write a test case with the *T_TEST_CASE()* macro followed by a function body:

```
1  T_TEST_CASE(name)
2  {
3      /* Your test case code */
4  }
```

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 constructors.

Listing 6.1: Test Case Example

```
1   #include <t.h>
2
3   static int add(int a, int b)
4   {
5       return a + b;
6   }
7
8   T_TEST_CASE(a_test_case)
9   {
10      int actual_value;
11
12      actual_value = add(1, 1);
13      T_eq_int(actual_value, 2);
14      T_true(false, "a test failure message");
15  }
```

Listing 6.2: Test Case Report

```
1  B:a_test_case
2  P:0:8:UI1:test-simple.c:13
3  F:1:8:UI1:test-simple.c:14:a test failure message
4  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 72).

6.1.3 Test Fixture

You can write a test case with a test fixture with the *T_TEST_CASE_FIXTURE()* macro followed by a function body:

```
1  T_TEST_CASE_FIXTURE(name, fixture)
2  {
3      /* Your test case code */
4  }
```

6.1. The RTEMS Test Framework

49

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*. The test fixture provides methods to setup, stop and tear down a test case. A context is passed to the methods. The initial context is defined by the read-only fixture object. The context can be obtained by the *T_fixture_context()* function. It can be set within the scope of one test case by the *T_set_fixture_context()* function. This can be used for example to dynamically allocate a test environment in the setup method.

Listing 6.3: Test Fixture Example

```c
#include <t.h>

static int initial_value = 3;

static int counter;

static void
setup(void *ctx)
{
    int *c;

    T_log(T_QUIET, "setup begin");
    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,
```

```
48        .stop = stop,
49        .teardown = teardown,
50        .initial_context = &initial_value
51 };
52
53 T_TEST_CASE_FIXTURE(fixture, &fixture)
54 {
55        T_assert_true(true, "all right");
56        T_assert_true(false, "test fails and we stop the test case");
57        T_log(T_QUIET, "not reached");
58 }
```

Listing 6.4: Test Fixture Report

```
1  B:fixture
2  L:setup begin
3  P:0:0:UI1:test-fixture.c:13
4  P:1:0:UI1:test-fixture.c:14
5  P:2:0:UI1:test-fixture.c:18
6  L:setup end
7  P:3:0:UI1:test-fixture.c:55
8  F:4:0:UI1:test-fixture.c:56:test fails and we stop the test case
9  L:stop begin
10 P:5:0:UI1:test-fixture.c:28
11 L:stop end
12 L:teardown begin
13 P:6:0:UI1:test-fixture.c:40
14 P:7:0:UI1:test-fixture.c:42
15 L:teardown end
16 E:fixture:N:8:F:1
```

6.1.4 Test Case Planning

Each 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.

```
1 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 6.5: Test Planning Example

```
1 #include <t.h>
2
3 T_TEST_CASE(wrong_step)
4 {
5        T_plan(2);
6        T_step_true(0, true, "all right");
7        T_step_true(2, true, "wrong step");
8 }
```

```
9
10  T_TEST_CASE(plan_ok)
11  {
12      T_plan(1);
13      T_step_true(0, true, "all right");
14  }
15
16  T_TEST_CASE(plan_failed)
17  {
18      T_plan(2);
19      T_step_true(0, true, "not enough steps");
20      T_quiet_true(true, "quiet test do not count");
21  }
22
23  T_TEST_CASE(double_plan)
24  {
25      T_plan(99);
26      T_plan(2);
27  }
28
29  T_TEST_CASE(steps)
30  {
31      T_step(0, "a");
32      T_plan(3);
33      T_step(1, "b");
34      T_step(2, "c");
35  }
```

Listing 6.6: Test Planning Report

```
1   B:wrong_step
2   P:0:0:UI1:test-plan.c:6
3   F:1:0:UI1:test-plan.c:7:planned step (2)
4   E:wrong_step:N:2:F:1
5   B:plan_ok
6   P:0:0:UI1:test-plan.c:13
7   E:plan_ok:N:1:F:0
8   B:plan_failed
9   P:0:0:UI1:test-plan.c:19
10  F:*:0:UI1:*:*:actual steps (1), planned steps (2)
11  E:plan_failed:N:1:F:1
12  B:double_plan
13  F:*:0:UI1:*:*:planned steps (99) already set
14  E:double_plan:N:0:F:1
15  B:steps
16  P:0:0:UI1:test-plan.c:31
17  P:1:0:UI1:test-plan.c:33
18  P:2:0:UI1:test-plan.c:34
19  E:steps:N:3:F:0
```

6.1.5 Test Case Resource Accounting

The framework can check if various resources are 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 6.7: 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 6.8: 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
```

6.1. The RTEMS Test Framework

6.1.6 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 6.9: 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);
    T_step_eq_int(1, *i, 0);
    *i = 1;
}
```

Listing 6.10: Test Case Scoped Dynamic Memory Report

```
1  B:malloc_free
2  P:0:0:UI1:test-malloc.c:8
3  E:malloc_free:N:1:F:0:D:0.005200
4  B:malloc_auto
5  P:0:0:UI1:test-malloc.c:17
6  E:malloc_auto:N:1:F:0:D:0.004790
7  B:zalloc_auto
8  P:0:0:UI1:test-malloc.c:35
9  P:1:0:UI1:test-malloc.c:36
10 P:2:0:UI1:test-malloc.c:26
11 E:zalloc_auto:N:3:F:0:D:0.006583
```

6.1.7 Test Case Destructors

You can add test case destructors with *T_add_destructor()*. They 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. Do not use stack memory or dynamic memory obtained via *T_malloc()*, *T_calloc()* or *T_zalloc()* for the *T_destructor* structure.

```
1  void T_add_destructor(T_destructor *destructor,
2      void (*destroy)(T_destructor *));
3
4  void T_remove_destructor(T_destructor *destructor);
```

Listing 6.11: Test Case Destructor Example

```
1  #include <t.h>
2
3  static void
4  destroy(T_destructor *dtor)
5  {
6      (void)dtor;
7      T_step(0, "destroy");
8  }
9
10 T_TEST_CASE(destructor)
11 {
12     static T_destructor dtor;
13
14     T_plan(1);
15     T_add_destructor(&dtor, destroy);
16 }
```

Listing 6.12: Test Case Destructor Report

```
1  B:destructor
2  P:0:0:UI1:test-destructor.c:7
3  E:destructor:N:1:F:0:D:0.003714
```

6.1.8 Test Checks

A *test check* determines if the actual value presented to the test check meets its expectation. The actual value should represent the outcome of a test action. If the actual value is all right, then the test check passes, otherwise the 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.

6.1.8.1 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.

6.1.8.2 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.

6.1.8.3 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.

The following names for test check type variants 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*.

6.1.8.4 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_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, ...);
```

The message is only printed in case the test check fails. The format parameter is mandatory.

Listing 6.13: Boolean Test Checks Example

```
1  #include <t.h>
2
3  T_TEST_CASE(example)
4  {
5      T_true(true, "test passes, no message output");
6      T_true(false, "test fails");
7      T_quiet_true(true, "quiet test passes, no output at all");
8      T_quiet_true(false, "quiet test fails");
9      T_step_true(2, true, "step test passes, no message output");
10     T_step_true(3, false, "step test fails");
11     T_assert_false(true, "this is a format %s", "string");
12 }
```

Listing 6.14: Boolean Test Checks Report

```
1  B:example
2  P:0:0:UI1:test-example.c:5
3  F:1:0:UI1:test-example.c:6:test fails
4  F:*:0:UI1:test-example.c:8:quiet test fails
5  P:2:0:UI1:test-example.c:9
6  F:3:0:UI1:test-example.c:10:step test fails
7  F:4:0:UI1:test-example.c:11:this is a format string
8  E:example:N:5:F:4
```

6.1.8.5 Generic Types

The following test checks for data types with an equality (==) or inequality (!=) operator are available:

```
1  void T_eq(T a, T e, const char *fmt, ...);
2  void T_assert_eq(T a, T e, const char *fmt, ...);
3  void T_quiet_eq(T a, T e, const char *fmt, ...);
4  void T_step_eq(unsigned int step, T a, T e, const char *fmt, ...);
5  void T_step_assert_eq(unsigned int step, T a, T e, const char *fmt, ...);
6
7  void T_ne(T a, T e, const char *fmt, ...);
8  void T_assert_ne(T a, T e, const char *fmt, ...);
9  void T_quiet_ne(T a, T e, const char *fmt, ...);
10 void T_step_ne(unsigned int step, T a, T e, const char *fmt, ...);
11 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.

6.1.8.6 Pointers

The following test checks for pointers are available:

```
1  void T_eq_ptr(const void *a, const void *e);
2  void T_assert_eq_ptr(const void *a, const void *e);
3  void T_quiet_eq_ptr(const void *a, const void *e);
4  void T_step_eq_ptr(unsigned int step, const void *a, const void *e);
```

```
5 void T_step_assert_eq_ptr(unsigned int step, const void *a, const void *e);
6
7 void T_ne_ptr(const void *a, const void *e);
8 void T_assert_ne_ptr(const void *a, const void *e);
9 void T_quiet_ne_ptr(const void *a, const void *e);
10 void T_step_ne_ptr(unsigned int step, const void *a, const void *e);
11 void T_step_assert_ne_ptr(unsigned int step, const void *a, const void *e);
12
13 void T_null(const void *a);
14 void T_assert_null(const void *a);
15 void T_quiet_null(const void *a);
16 void T_step_null(unsigned int step, const void *a);
17 void T_step_assert_null(unsigned int step, const void *a);
18
19 void T_not_null(const void *a);
20 void T_assert_not_null(const void *a);
21 void T_quiet_not_null(const void *a);
22 void T_step_not_null(unsigned int step, const void *a);
23 void T_step_assert_not_null(unsigned int step, const void *a);
```

An automatically generated message is printed in case the test check fails.

6.1.8.7 Memory Areas

The following test checks for memory areas are available:

```
1 void T_eq_mem(const void *a, const void *e, size_t n);
2 void T_assert_eq_mem(const void *a, const void *e, size_t n);
3 void T_quiet_eq_mem(const void *a, const void *e, size_t n);
4 void T_step_eq_mem(unsigned int step, const void *a, const void *e, size_t n);
5 void T_step_assert_eq_mem(unsigned int step, const void *a, const void *e, size_t n);
6
7 void T_ne_mem(const void *a, const void *e, size_t n);
8 void T_assert_ne_mem(const void *a, const void *e, size_t n);
9 void T_quiet_ne_mem(const void *a, const void *e, size_t n);
10 void T_step_ne_mem(unsigned int step, const void *a, const void *e, size_t n);
11 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.

6.1.8.8 Strings

The following test checks for strings are available:

```
1 void T_eq_str(const char *a, const char *e);
2 void T_assert_eq_str(const char *a, const char *e);
3 void T_quiet_eq_str(const char *a, const char *e);
4 void T_step_eq_str(unsigned int step, const char *a, const char *e);
5 void T_step_assert_eq_str(unsigned int step, const char *a, const char *e);
6
7 void T_ne_str(const char *a, const char *e);
8 void T_assert_ne_str(const char *a, const char *e);
9 void T_quiet_ne_str(const char *a, const char *e);
```

```
10  void T_step_ne_str(unsigned int step, const char *a, const char *e);
11  void T_step_assert_ne_str(unsigned int step, const char *a, const char *e);
12
13  void T_eq_nstr(const char *a, const char *e, size_t n);
14  void T_assert_eq_nstr(const char *a, const char *e, size_t n);
15  void T_quiet_eq_nstr(const char *a, const char *e, size_t n);
16  void T_step_eq_nstr(unsigned int step, const char *a, const char *e, size_t n);
17  void T_step_assert_eq_nstr(unsigned int step, const char *a, const char *e, size_t n);
18
19  void T_ne_nstr(const char *a, const char *e, size_t n);
20  void T_assert_ne_nstr(const char *a, const char *e, size_t n);
21  void T_quiet_ne_nstr(const char *a, const char *e, size_t n);
22  void T_step_ne_nstr(unsigned int step, const char *a, const char *e, size_t n);
23  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.

6.1.8.9 Characters

The following test checks for characters (*char*) are available:

```
1   void T_eq_char(char a, char e);
2   void T_assert_eq_char(char a, char e);
3   void T_quiet_eq_char(char a, char e);
4   void T_step_eq_char(unsigned int step, char a, char e);
5   void T_step_assert_eq_char(unsigned int step, char a, char e);
6
7   void T_ne_char(char a, char e);
8   void T_assert_ne_char(char a, char e);
9   void T_quiet_ne_char(char a, char e);
10  void T_step_ne_char(unsigned int step, char a, char e);
11  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.

6.1.8.10 Integers

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);
9   void T_quiet_ne_xyz(I a, I e);
10  void T_step_ne_xyz(unsigned int step, I a, I e);
11  void T_step_assert_ne_xyz(unsigned int step, I a, I e);
12
13  void T_ge_xyz(I a, I e);
14  void T_assert_ge_xyz(I a, I e);
```

```
15  void T_quiet_ge_xyz(I a, I e);
16  void T_step_ge_xyz(unsigned int step, I a, I e);
17  void T_step_assert_ge_xyz(unsigned int step, I a, I e);
18
19  void T_gt_xyz(I a, I e);
20  void T_assert_gt_xyz(I a, I e);
21  void T_quiet_gt_xyz(I a, I e);
22  void T_step_gt_xyz(unsigned int step, I a, I e);
23  void T_step_assert_gt_xyz(unsigned int step, I a, I e);
24
25  void T_le_xyz(I a, I e);
26  void T_assert_le_xyz(I a, I e);
27  void T_quiet_le_xyz(I a, I e);
28  void T_step_le_xyz(unsigned int step, I a, I e);
29  void T_step_assert_le_xyz(unsigned int step, I a, I e);
30
31  void T_lt_xyz(I a, I e);
32  void T_assert_lt_xyz(I a, I e);
33  void T_quiet_lt_xyz(I a, I e);
34  void T_step_lt_xyz(unsigned int step, I a, I e);
35  void T_step_assert_lt_xyz(unsigned int step, I a, I e);
```

The type variant *xyz* must be *schar, uchar, short, ushort, int, uint, long, ulong, ll, ull, i8, u8, i16, u16, i32, u32, i64, u64, iptr, uptr, ssz,* or *sz.*

The type name *I* must be compatible to the type variant.

An automatically generated message is printed in case the test check fails.

6.1.8.11 RTEMS Status Codes

The following test checks for RTEMS status codes are available:

```
1   void T_rsc(rtems_status_code a, rtems_status_code e);
2   void T_assert_rsc(rtems_status_code a, rtems_status_code e);
3   void T_quiet_rsc(rtems_status_code a, rtems_status_code e);
4   void T_step_rsc(unsigned int step, rtems_status_code a, rtems_status_code e);
5   void T_step_assert_rsc(unsigned int step, rtems_status_code a, rtems_status_code e);
6
7   void T_rsc_success(rtems_status_code a);
8   void T_assert_rsc_success(rtems_status_code a);
9   void T_quiet_rsc_success(rtems_status_code a);
10  void T_step_rsc_success(unsigned int step, rtems_status_code a);
11  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.

6.1.8.12 POSIX Error Numbers

The following test checks for POSIX error numbers are available:

```
1   void T_eno(int a, int e);
2   void T_assert_eno(int a, int e);
3   void T_quiet_eno(int a, int e);
4   void T_step_eno(unsigned int step, int a, int e);
```

```
5  void T_step_assert_eno(unsigned int step, int a, int e);
6
7  void T_eno_success(int a);
8  void T_assert_eno_success(int a);
9  void T_quiet_eno_success(int a);
10 void T_step_eno_success(unsigned int step, int a);
11 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.

6.1.8.13 POSIX Status Codes

The following test checks for POSIX status codes are available:

```
1  void T_psx_error(int a, int eno);
2  void T_assert_psx_error(int a, int eno);
3  void T_quiet_psx_error(int a, int eno);
4  void T_step_psx_error(unsigned int step, int a, int eno);
5  void T_step_assert_psx_error(unsigned int step, int a, int eno);
6
7  void T_psx_success(int a);
8  void T_assert_psx_success(int a);
9  void T_quiet_psx_success(int a);
10 void T_step_psx_success(unsigned int step, int a);
11 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 6.15: POSIX Status Code Example

```
1  #include <t.h>
2
3  #include <sys/stat.h>
4  #include <errno.h>
5
6  T_TEST_CASE(stat)
7  {
8      struct stat st;
9      int status;
10
11     errno = 0;
12     status = stat("foobar", &st);
13     T_psx_error(status, ENOENT);
14 }
```

Listing 6.16: POSIX Status Code Report

```
1 B:stat
2 P:0:0:UI1:test-psx.c:13
3 E:stat:N:1:F:0
```

6.1.9 Custom Log Messages

You can print custom log messages with the *T_log()* function:

```
1 void T_log(T_verbosity verbosity, char const *fmt, ...);
```

A newline is automatically added to terminate the log message line.

Listing 6.17: Custom Log Message Example

```
1 #include <t.h>
2
3 T_TEST_CASE(log)
4 {
5     T_log(T_NORMAL, "a custom message %i, %i, %i", 1, 2, 3);
6     T_set_verbosity(T_QUIET);
7     T_log(T_NORMAL, "not verbose enough");
8 }
```

Listing 6.18: Custom Log Message Report

```
1 B:log
2 L:a custom message 1, 2, 3
3 E:log:N:0:F:0
```

6.1.10 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*.

```
1 T_ticks T_tick(void);
2
3 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.

```
1 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.

```
1  T_time T_ticks_to_time(T_ticks ticks);
2
3  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.

```
1  T_time T_seconds_and_nanoseconds_to_time(uint32_t s, uint32_t ns);
2
3  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 representation. The time unit of the string representation is seconds. The precision of the string representation may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (*T_time_string*).

```
1  const char *T_time_to_string_ns(T_time time, T_time_string buffer);
2
3  const char *T_time_to_string_us(T_time time, T_time_string buffer);
4
5  const char *T_time_to_string_ms(T_time time, T_time_string buffer);
6
7  const char *T_time_to_string_s(T_time time, T_time_string buffer);
```

Listing 6.19: Time String Example

```
1  #include <t.h>
2
3  T_TEST_CASE(time_to_string)
4  {
5      T_time_string ts;
6      T_time t;
7      uint32_t s;
8      uint32_t ns;
9
10     t = T_seconds_and_nanoseconds_to_time(0, 123456789);
11     T_eq_str(T_time_to_string_ns(t, ts), "0.123456789");
12     T_eq_str(T_time_to_string_us(t, ts), "0.123456");
13     T_eq_str(T_time_to_string_ms(t, ts), "0.123");
14     T_eq_str(T_time_to_string_s(t, ts), "0");
15
16     T_time_to_seconds_and_nanoseconds(t, &s, &ns);
17     T_eq_u32(s, 0);
18     T_eq_u32(ns, 123456789);
19 }
```

Listing 6.20: Time String Report

```
1  B:time_to_string
2  P:0:0:UI1:test-time.c:11
3  P:1:0:UI1:test-time.c:12
4  P:2:0:UI1:test-time.c:13
5  P:3:0:UI1:test-time.c:14
6  P:4:0:UI1:test-time.c:17
7  P:5:0:UI1:test-time.c:18
8  E:time_to_string:N:6:F:0:D:0.005250
```

You can convert a tick value into a string representation. The time unit of the string representation is seconds. The precision of the string representation may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (*T_time_string*).

```c
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);
```

6.1.11 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.

```c
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);

void T_measure_runtime(T_measure_runtime_context *ctx,
    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_VALID_CACHE
  Disable the *ValidCache* 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:

ValidCache
  Before the *body* request handler is invoked a memory area with twice the size of the outer-most data cache is completely read. This fills the data cache with valid cache lines which are unrelated to the *body* request handler.

  You can disable this variant with the *T_MEASURE_RUNTIME_DISABLE_VALID_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
  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 (*M:L*). The active workers range from one up 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 6.21: Code Runtime Measurement Example

```c
#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 6.22: Code Runtime Measurement Report

```
B:measure_empty
P:0:0:UI1:test-rtems-measure.c:18
M:B:Empty
M:V:ValidCache
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
```

```
24  M:E:Empty:D:0.000017046
25  M:B:Empty
26  M:V:DirtyCache
27  M:N:1024
28  M:MI:0.000000007
29  M:Q1:0.000000007
30  M:Q2:0.000000007
31  M:Q3:0.000000008
32  M:MX:0.000000559
33  M:MAD:0.000000000
34  M:D:0.000033244
35  M:E:Empty:D:1.887834875
36  M:B:Empty
37  M:V:Load
38  M:L:1
39  M:N:1024
40  M:MI:0.000000000
41  M:Q1:0.000000002
42  M:Q2:0.000000002
43  M:Q3:0.000000003
44  M:MX:0.000000288
45  M:MAD:0.000000000
46  M:D:0.000002421
47  M:E:Empty:D:0.001798809
48  [... 22 more load variants ...]
49  M:E:Empty:D:0.021252583
50  M:B:Empty
51  M:V:Load
52  M:L:24
53  M:N:1024
54  M:MI:0.000000001
55  M:Q1:0.000000002
56  M:Q2:0.000000002
57  M:Q3:0.000000003
58  M:MX:0.000001183
59  M:MAD:0.000000000
60  M:D:0.000003406
61  M:E:Empty:D:0.015188063
62  E:measure_empty:N:1:F:0:D:14.284869
```

6.1.12 Test Runner

You can call the *T_main()* function to run all registered test cases.

```
1  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:

```
1  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.

```
1 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()*.

```
1  void T_run_initialize(const T_config *config);
2
3  void T_run_all(void);
4
5  void T_run_by_name(const char *name);
6
7  void T_case_begin(const char *name, const T_fixture *fixture);
8
9  void T_case_end(void);
10
11 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.

```
1  typedef enum {
2      T_EVENT_RUN_INITIALIZE,
3      T_EVENT_CASE_EARLY,
4      T_EVENT_CASE_BEGIN,
5      T_EVENT_CASE_END,
6      T_EVENT_CASE_LATE,
7      T_EVENT_RUN_FINALIZE
8  } T_event;
9
10 typedef void (*T_action)(T_event, const char *);
11
12 typedef void (*T_putchar)(int, void *);
13
14 typedef struct {
15     const char *name;
16     T_putchar putchar;
17     void *putchar_arg;
18     T_verbosity verbosity;
19     T_time (*now)(void);
20     size_t action_count;
21     const T_action *actions;
22 } 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. 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*).

6.1.13 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:

```
1  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:

```
1  A:xyz
2  S:Platform:RTEMS
3  [...]
4  E:a:N:2:F:1
5  E:b:N:0:F:1
6  E:c:N:1:F:1
7  E:d:N:6:F:0
8  Z:xyz:C:4:N:9:F:3
```

The same example with *T_NORMAL* verbosity:

```
1   A:xyz
2   S:Platform:RTEMS
3   [...]
4   B:a
5   F:1:0:UI1:test-verbosity.c:6:test fails
6   E:a:N:2:F:1
7   B:b
8   F:*:0:UI1:test-verbosity.c:12:quiet test fails
9   E:b:N:0:F:1
10  B:c
11  F:0:0:UI1:test-verbosity.c:17:this is a format string
12  E:c:N:1:F:1
13  B:d
14  E:d:N:6:F:0
15  Z:xyz:C:4:N:9:F:3
```

The same example with *T_VERBOSE* verbosity:

```
 1 A:xyz
 2 S:Platform:RTEMS
 3 [...]
 4 B:a
 5 P:0:0:UI1:test-verbosity.c:5
 6 F:1:0:UI1:test-verbosity.c:6:test fails
 7 E:a:N:2:F:1
 8 B:b
 9 F:*:0:UI1:test-verbosity.c:12:quiet test fails
10 E:b:N:0:F:1
11 B:c
12 F:0:0:UI1:test-verbosity.c:17:this is a format string
13 E:c:N:1:F:1
14 B:d
15 P:0:0:UI1:test-verbosity.c:22
16 P:1:0:UI1:test-verbosity.c:23
17 P:2:0:UI1:test-verbosity.c:24
18 P:3:0:UI1:test-verbosity.c:25
19 P:4:0:UI1:test-verbosity.c:26
20 P:5:0:UI1:test-verbosity.c:27
21 E:d:N:6:F:0
22 Z:xyz:C:4:N:9:F:3
```

6.1.14 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:L:<Load>

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 **ValidCache**, **HotCache**, **DirtyCache**, or **Load**.

<Load>
The active load workers count 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 6.23: Example Test Report

```
1 A:xyz
2 S:Platform:RTEMS
3 S:Compiler:7.4.0 20181206 (RTEMS 5, RSB e0aec65182449a4e22b820e773087636edaf5b32, Newlib↵
  ↪1d35a003f)
4 S:Version:5.0.0.820977c5af17c1ca2f79800d64bd87ce70a24c68
```

```
 5  S:BSP:erc32
 6  S:RTEMS_DEBUG:1
 7  S:RTEMS_MULTIPROCESSING:0
 8  S:RTEMS_POSIX_API:1
 9  S:RTEMS_PROFILING:0
10  S:RTEMS_SMP:1
11  B:timer
12  P:0:0:UI1:test-rtems.c:26
13  P:1:0:UI1:test-rtems.c:29
14  P:2:0:UI1:test-rtems.c:33
15  P:3:0:ISR:test-rtems.c:14
16  P:4:0:ISR:test-rtems.c:15
17  P:5:0:UI1:test-rtems.c:38
18  P:6:0:UI1:test-rtems.c:39
19  P:7:0:UI1:test-rtems.c:42
20  E:timer:N:8:F:0:D:0.019373
21  B:rsc_success
22  P:0:0:UI1:test-rtems.c:59
23  F:1:0:UI1:test-rtems.c:60:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
24  F:*:0:UI1:test-rtems.c:62:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
25  P:2:0:UI1:test-rtems.c:63
26  F:3:0:UI1:test-rtems.c:64:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
27  E:rsc_success:N:4:F:3:D:0.011128
28  B:rsc
29  P:0:0:UI1:test-rtems.c:48
30  F:1:0:UI1:test-rtems.c:49:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
31  F:*:0:UI1:test-rtems.c:51:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
32  P:2:0:UI1:test-rtems.c:52
33  F:3:0:UI1:test-rtems.c:53:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
34  E:rsc:N:4:F:3:D:0.011083
35  Z:xyz:C:3:N:16:F:6:D:0.047201
36  Y:ReportHash:SHA256:e5857c520dd9c9b7c15d4a76d78c21ccc46619c30a869ecd11bbcd1885155e0b
```

6.1.15 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.

6.1.16 Supported Platforms

The framework runs on FreeBSD, MSYS2, Linux and RTEMS.

6.2 Test Framework Requirements for RTEMS

The requirements on a test framework suitable for RTEMS are:

6.2.1 License Requirements

TF.License.Permissive
  The test framework shall have a permissive open source license such as BSD-2-Clause.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 Performance Requirements

TF.Performance.RTEMS.No64BitDivision
  The test framework shall not use 64-bit divisions on RTEMS.

6.3 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*).

6.3.1 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*.

6.3.2 CBDD

The CBDD framework uses the C blocks extension from clang. This violates *TF.Portability.C11*.

6.3.3 Google Test

Google Test 1.8.1 is supported by RTEMS. Unfortunately, it is written in C++ and is to heavy weight for low-end platforms. Otherwise it is a nice framework.

6.3.4 Unity

The Unity Test API does not meet our requirements. There was a discussion on the mailing list in 2013.

6.4 Standard Test Report Formats

6.4.1 JUnit XML

A common test report format is JUnit XML.

```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 48) to the JUnit XML format.

6.4.2 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 48) to the TAP format.

SOFTWARE RELEASE MANAGEMENT

TBD write content

7.1 Software Change Report Generation

TBD - What goes here?

7.2 Version Description Document (VDD) Generation

TBD - discuss how generated. Preferably Dannie's project

This URL may be of use but it probably Trac auto-generated and can only be referenced: https://devel.rtems.org/wiki/TracChangeLog

USER'S MANUALS

TBD - write and link to useful documentation, potential URLs:

Reference the RTEMS Classic API Guide

- https://docs.rtems.org/doc-current/share/rtems/pdf/c_user.pdf

Reference any other existing user documentation

- https://docs.rtems.org/doxygen/cpukit/html/index.html

- https://devel.rtems.org/

- http://www.rtems.com/

- https://www.rtems.org/onlinedocs.html

- https://devel.rtems.org/wiki/Developer/Contributing

- https://docs.rtems.org/releases/rtemsdocs-4.10.1/share/rtems/html/

8.1 Documentation Style Guidelines

TBD - write me

LICENSING REQUIREMENTS

All artifacts shall adhere to RTEMS Project licensing requirements. Currently, the preferred licenses are CC-BY-SA-4.0 license for documentation and "Two Paragraph BSD" for source code.

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://devel.rtems.org/ticket/3053 for details.

TBD - Convert the following to Rest and insert into this file TBD - https://devel.rtems.org/wiki/Developer/Coding/Conventions#Licenses

TBD - Review and make sure this includes info on BSD variants

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 10.1: Table 1. Core Qualification Artifacts

Category	Artifact	Intent
Requirements	Software Requirements Specification (SRS) Requirements Management	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 / Verifica-	The project shall record, address, and track to closure the results of software verification activities.

Chapter 10. Appendix: Core Qualification Artifacts/Documents

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: