Realm Management Monitor

TF-RMM Contributors

Apr 18, 2024

CONTENTS

CHAPTER ONE

ABOUT

1.1 Readme for TF-RMM

TF-RMM (or simply RMM) is the Trusted Firmware Implementation of the Realm Management Monitor (RMM) Specification. The RMM is a software component that runs at Realm EL2 and forms part of a system which implements the Arm Confidential Compute Architecture (Arm CCA). Arm CCA is an architecture which provides Protected Execution Environments called Realms.

Prior to Arm CCA, virtual machines have to trust hypervisors that manage them and a resource that is managed by the hypervisor is also accessible by it. Exploits against the hypervisors can leak confidential data held in the virtual machines. Arm CCA introduces a new confidential compute environment called a *Realm*. Any code or data belonging to a *Realm*, whether in memory or in registers, cannot be accessed or modified by the hypervisor. This means that the Realm owner does not need to trust the hypervisor that manages the resources used by the Realm.

The Realm VM is initiated and controlled by the Normal world Hypervisor. To allow the isolated execution of the Realm VM, a new component called the Realm Management Monitor (RMM) is introduced, executing at R_EL2. The hypervisor interacts with the RMM via Realm Management Interface (RMI) to manage the Realm VM. Policy decisions, such as which Realm to run or what memory to be delegated to the Realm are made by the hypervisor and communicated via the RMI. The RMM also provides services to the Realm via the Realm Service Interface (RSI). These services include cryptographic services and attestation. The Realm initial state can be measured and an attestation report, which also includes platform attestation, can be requested via RSI. The RSI is also the channel for memory management requests from the Realm VM to the RMM.

The following diagram shows the complete Arm CCA software stack running a confidential Realm VM :

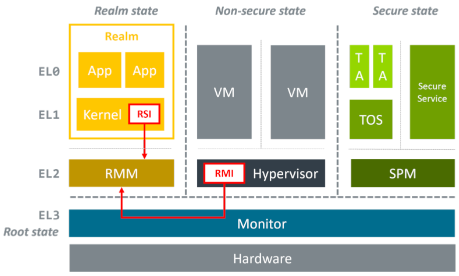


Figure 1. Realm VM execution

The TF-RMM interacts with the Root EL3 Firmware via the RMM-EL3 Communication Interface and this is implemented by the reference EL3 Firmware implementation TF-A.

More details about the RMM and how it fits in the Software Stack can be found in Arm CCA Software Stack Guide.

The Change-log and Release notes has the details of features implemented by this version of TF-RMM and lists any known issues.

1.1.1 License

Unless specifically indicated otherwise in a file, TF-RMM files are provided under the BSD-3-Clause License. For contributions, please see License and Copyright for Contributions.

Third Party Projects

The TF-RMM project requires to be linked with certain other 3rd party projects and they are to be cloned from their repositories into `ext` folder before building. The projects are MbedTLS, t_cose, QCBOR and CppUTest.

The project also contains files which are imported from other projects into the source tree and may have a different license. Such files with different licenses are listed in the table below. This table is used by the `checkspdx` tool in the project to verify license headers.

Table 1: **List of files with different license**

File	License
lib/libc/src/printf.c	MIT
lib/libc/include/stdio.h	MIT
lib/libc/src/strlcpy.c	ISC
lib/libc/src/strnlen.c	BSD-2-Clause
lib/allocator/src/memory_alloc.c	Apache-2.0

1.1.2 Contributing

We gratefully accept bug reports and contributions from the community. Please see the Contributor's Guide for details on how to do this.

1.1.3 Feedback and support

Feedback is requested via email to: tf-rmm@lists.trustedfirmware.org.

To report a bug, please file an issue on Github

1.2 Project Maintenance

Realm Management Monitor (RMM) is an open governance community project. All contributions are ultimately merged by the maintainers listed below. Technical ownership of most parts of the codebase falls on the code owners listed below. An acknowledgement from these code owners is required before the maintainers merge a contribution.

More details may be found in the Project Maintenance Process document.

1.2.1 Maintainers

Mail
Alexei Fedorov <Alexei.Fedorov@arm.com>

GitHub ID
AlexeiFedorov

Mail
Arunachalam Ganapathy <arunachalam.ganapathy@arm.com>

GitHub ID
arugan02

Mail
Dan Handley <dan.handley@arm.com>

GitHub ID
danh-arm

Mail
Javier Almansa Sobrino <javier.almansasobrino@arm.com>

GitHub ID
javier-almansasobrino

Mail
Mate Toth-Pal <mate.toth-pal@arm.com>

GitHub ID
Máté Tóth-Pál

Mail
Soby Mathew <soby.mathew@arm.com>

GitHub ID
soby-mathew

1.3 Change-log and Release notes

1.3.1 v0.4.0

The following sections have the details on the release. This release has been verified with TF-A v2.10 release.

New features in this release

- Added initial partial support for analysing RMM source code with CBMC (https://www.cprover.org/cbmc/).
  - A new HOST_VARIANT, *host_cbmc*, has been introduced for this purpose.
  - The CBMC testbench files and autogenerated files from RMM machine readable specification are imported into the source tree.
  - An application note for the same is added to the documentation.
- Aligned the implementation to RMM v1.0 EAC5 specification.
  - The relevant tag for the alignment is rmm-spec-v1.0-eac5.
  - There is also an intermediate RMM v1.0 EAC2 alignment which is tagged rmm-spec-v1.0-eac2.

- Supported save and restore of Non Secure SME context when Realms are scheduled.
  - The SIMD abstraction in RMM was reworked to cater for this requirement.
  - Added support to emulate SME specific feature ID registers.
  - Support injecting UNDEF exception into realm when SME is accessed within it.
  - Also RMM now can handle SVE hint bit as specified by SMCCC v1.3 specification.
- Added TF-RMM Threat Model to the documentation.
- Added capability to privately map the per-CPU stack.
  - This contains any stack overflows to the particular CPU and prevents a CPU from corrupting another CPU stack.
- Added FEAT_PAUTH and FEAT_BTI support to RMM and also capability to use FEAT_PAUTH within realms.
- Migrate to PSA Crypto API for attestation and measurement functionality in RMM.
- Added FEAT_LPA2 support to Stage 1 MMU code (lib/xlat) in RMM.
- Added Stage 1 MMU setup design document.

Build/Testing/Tooling improvements

- Added static commit message checker which enforces the commit message guidelines mandated for the project.
- Added clang-tidy checker as one of the static analyzers.
  - Several fixes to errors flagged by the static checker have been fixed.
- Fixed issues found in xlat lib unittests.
- Added github workflow for git submodules so that the TF-RMM dependencies display correctly in github.
- Added github workflow to configure an automatic message for PRs on GitHub and also build and run RMM unittests for every update of the *main* branch.
- Added FEAT_LPA2 unit tests for lib/xlat module.
- Added RSI logger unit tests.

Platforms

- The support for QEMU virt platform was merged.

Bug fixes/improvements in this release

- Fixed issue with TLB invalidations for unprotected mappings during RMI_RTT_DESTROY command.
- Fixed an issue wherein attest token write may return without releasing lock on the last level RTT of the mapped buffer.
- Enable TSW bit in hcr_el2 when executing in Realm world so as to trap any data cache maintenance instructions that operate by Set/Way.
- Fixed issues flagged by coverity online scan. The defects detected can be found in the TF-RMM coverity scan online homepage.
- Fixed issues in s2tt management related to NS memory assignment/unassignment.

- Added missing check to gicv3_hcr field.
- Cache line align xlat lib data structures accessed by secondary CPUs to avoid data corruption due to mismatched memory attribute accesses by RMM during warm boot.
- Corrected linker options when building qcbor library.
- Fixes to comply with MISRA coding guidelines.
- Adjusted mbedTLS heap size depending on MAX_CPUS in RMM.
- Fixed issue with RMI_DATA_CREATE_UNKNOWN setting RIPAS to RAM.
- Added 'ipa_bound' failure condition in RMI_DATA_DESTROY handler. Also added 'level_bound' failure condition for RMI_RTT_MAP_UNPROTECTED and RMI_RTT_UNMAP_UNPROTECTED command handlers.
- Fixed issue with rsi_log_on_exit() and modified the logging format.
- Fixed issue with change *ipa_align* failure condition.
- Unified design of RSI/PSCI handlers.
- The issue with RMM config `RMM_FPU_USE_AT_REL2` is fixed and the SIMD registers are saved and restored depending on the live register context in use which be one of FPU, SVE or SME.
- The compatibility check for RMM-EL3 interface version is hardened.
- Issue related to attestation token interruption flow is fixed.
- Enhanced the *fake_host* sample application to do Realm token creation.
- Fixed D-cache maintenance in fvp_set_dram_layout().
- Updated t_cose submodule to use upstream version rather than a forked version.

Known issues and limitations

- Some capabilities as mentioned in RMM v1.0 EAC5 specification are restricted or absent in TF-RMM as listed below:
  - The RMI_RTT_FOLD command only allows folding upto Level 2 even though the specification allows upto Level 1.
  - The support for Self-hosted debug in Realms is not implemented.
  - Although the RMM allows CCA attestation token sizes of larger than 4KB, there is a limitation on the size of the Platform attestation token part. On the RMM-EL3 interface, there is only a shared buffer of 4KB that is currently shared on the FVP. This needs to be enhanced so that larger platform token sizes can be tested.
- The *rmm-el3-ifc* component does not always reset the RMM to the correct state on encountering an error. This needs to be corrected.
- The invocation of mmio_emulation() and sea_inj() functions need to be mutually exclusive during schedule of a REC. Currently both the cases are allowed to be satisfied at the same time which is incorrect.

Upcoming features

- FEAT_LPA2 support for Stage 2 MMU code (s2tt) in RMM.
- Add unit-tests for Stage 2 MMU code (s2tt) and also any associated rework for the s2tt component.
- Enhance CBMC analysis to more RMI commands.
- Fuzz testing for RMM utilizing the *fake_host* architecture.
- Support for new capabilities like Device assignment as mandated by future versions of RMM specification.
- Integrate more static analyzers into RMM build system.
- Implement support for Self-hosted debug in realms.

1.3.2 v0.3.0

The following sections have the details on the release. This release has been verified with TF-A v2.9 release.

New features in this release

- Add support to create Realms which can make use of SVE, if present in hardware.
- Refactor the Stage 1 translation table library *lib/xlat* API to better fit RMM usage. Also harden dynamic mapping via slot buffer mechanism by use of `TRANSIENT` software defined attribute.
- Add PMU support for Realms as described by RMM v1.0 Beta0 specification.
- Support getting DRAM info from the Boot manifest dynamically at runtime.
  - RMM can now support the 2nd DDR bank on FVP.

Build/Testing improvements

- Define a unit test framework using CppUTest for RMM.
- Add unittests for *granule*, *slot-buffer* and Stage 1 translation table lib *xlat*.
- Improve the `fake-host` mock capability by adding support for per PE sysreg emulation.
- Improve the VA to PA mock layer for `fake-host`.
- Enable generation of gprof profiling data as part of `fake-host` runs.
- Improve the sample application on `host-build` platform by adding the cold attestation initialization flow. Also a sample minimal Realm create, run and destroy sequence is added to showcase the RMI calls involved.
- Further improvements to the unit test framework :
  - Restore the sysreg state between test runs so each test gets a known sysreg state.
  - Add capability to test assertions.
  - Support dynamic behaviour for test harness depending on requirement.
  - Add support for coverage report generation as part of unit test run.
- Build improvements in RMM:
  - Move mbedTLS build from configure stage to build stage.
  - Simplify QCBOR build.

  - Fix build artefact directory path to better cater to multi-config builds.

Bug fixes in this release

- Remove HVC exit handling from RMI_REC_ENTER handler.
- Fix parameter in measurement_extend_sha512().
- Fix issues in *lib/xlat* for some corner cases.
- Mask MTE capability from *id_aa64pfr1_el1* so that Realms can see that MTE is not supported.
- Add isb() after writes to *cptr_el2* system register.
- Fix the granule alignment check on granule_addr.
- Fix some cppcheck warnings.
- Properly handle errors for granule (un)delegate calls.
- Fix the incorrect bit map manipulation for tracking VMID for realms.
- Fix some incorrect Block mapping cases in Stage 2 translation.

Upcoming features

- RMM EAC Specification alignment.
- Support Self-Hosted Debug Realms.
- Support FEAT_PAuth for Realms and utilize the same for RMM.
- Support LPA2 for Stage 2 Realm translation tables.
- Threat model covering RMM data flows.
- Enable Bounded Model Checker (CBMC) for source analysis.
- Save and restore SME/SME2 context belonging to NS Host. This allows NS Host to make use of SME/SME2 when Realms are scheduled.

Known issues and limitations

- The size of `RsiHostCall` structure is 256 bytes in the implementation and aligns to RMM Beta1 specification rather than the 4 KB size specified in RMM Beta0 specification.
- The RMM Beta0 specification does not require to have a CBOR bytestream wrapper around the cca-platform-token and cca-realm-delegated-token, but the RMM implementation does so and this is aligned with later versions of the RMM specification (Beta2 onwards).
- The RMM config `RMM_FPU_USE_AT_REL2` does not work as intended and this config is disabled by default. This will be fixed in a future release.
- When the `RSI_ATTEST_TOKEN_CONTINUE` call is interrupted and then resumed later by Host via `RMI_REC_ENTER`, the original SMC is replayed again with the original arguments rather than returning `RSI_INCOMPLETE` error code to Realm. The result is that the interrupted RSI call is continued again till completion and then returns back to Realm with the appropriate error code.

1.3.3 v0.2.0

- This release has been verified with TF-A v2.8 release.
- The release has the following fixes and enhancements:
  - Add support to render documentation on read-the-docs.
  - Fix the known issue with RSI_IPA_STATE_GET returning `RSI_ERROR_INPUT` for a *destroyed* IPA instead of emulating data abort to NS Host.
  - Fix an issue with RSI_HOST_CALL not returning back to Host to emulate a stage2 data abort.
  - Harden an assertion check for `do_host_call()`.
- The other known issues and limitations remain the same as listed for *v0.1.0*.

1.3.4 v0.1.0

- First TF-RMM source release aligned to RMM Beta0 specification. The specified interfaces : Realm Management Interface (RMI) and Realm Service Interface (RSI) are implemented which can attest and run Realm VMs as described by the Arm CCA Architecture.

Upcoming features

- Support SVE, Self-Hosted Debug and PMU in Realms
- Support LPA2 for Stage 2 Realm translation tables.
- Threat model covering RMM data flows.
- Enable Bounded Model Checker (CBMC) for source analysis.
- Unit test framework based on *RMM Fake host architecture*.

Known issues and limitations

The following is a list of issues which are expected to be fixed in the future releases of TF-RMM :

- The size of `RsiHostCall` structure is 256 bytes in the implementation and aligns to RMM Beta1 specification rather than the 4 KB size specified in RMM Beta0 specification.
- The RSI_IPA_STATE_GET command returns error `RSI_ERROR_INPUT` for a *destroyed* IPA instead of emulating data abort to Host.
- The RMM Beta0 specification does not require to have a CBOR bytestream wrapper around the cca-platform-token and cca-realm-delegated-token, but the RMM implementation does so.

1.4 Developer Certificate of Origin

Developer Certificate of Origin Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 1 Letterman Drive Suite D4700 San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

1.5 License

BSD 3-Clause License

Copyright TF-RMM Contributors All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER

TWO

GETTING STARTED GUIDES

2.1 Prerequisite

This document describes the software requirements for building *RMM* for AArch64 target platforms.

It may possible to build *RMM* with combinations of software packages that are different from those listed below, however only the software described in this document can be officially supported.

2.2 Build Host

The *RMM* officially supports a limited set of build environments and setups. In this context, official support means that the environments listed below are actively used by team members and active developers, hence users should be able to recreate the same configurations by following the instructions described below. In case of problems, the *RMM* team provides support only for these environments, but building in other environments can still be possible.

We recommend at least Ubuntu 20.04 LTS (x64) for build environment. The arm64/AArch64 Ubuntu and other Linux distributions should also work fine, provided that the necessary tools and libraries can be installed.

2.3 Tool & Dependency overview

The following tools are required to obtain and build *RMM*:

Table 1: Tool dependencies

Name	Version	Component
C compiler	see *Setup Toolchain*	Firmware
CMake	>=3.15.0	Firmware, Documentation
GNU Make	>4.0	Firmware, Documentation
Python	3.x	Firmware, Documentation
Perl	>=5.26	Firmware, Documentation
ninja-build		Firmware (using Ninja Generator)
Sphinx	>=2.4,<3.0.0	Documentation
sphinxcontrib-plantuml		Documentation
sphinx-rtd-theme		Documentation
Git		Firmware, Documentation
Graphviz dot	>v2.38.0	Documentation
docutils	>v2.38.0	Documentation
gcovr	>=v4.2	Tools(Coverage analysis)
CBMC	>=5.84.0	Tools(CBMC analysis)
CPPcheck	>=1.90	Tools(CPPcheck)

2.4 Setup Toolchain

To compile *RMM* code for an AArch64 target, at least one of the supported AArch64 toolchains have to be available in the build environment.

Currently, the following compilers are supported:

- GCC (aarch64-none-elf-) >= 10.2-2020.11 (from the Arm Developer website)
- Clang+LLVM >= 14.0.0 (from the LLVM Releases website)

The respective compiler binary must be found in the shell's search path. Be sure to add the bin/ directory if you have downloaded a binary version. The toolchain to use can be set using `RMM_TOOLCHAIN` parameter and can be set to either *llvm* or *gnu*. The default toolchain is *gnu*.

For non-native AArch64 target build, the `CROSS_COMPILE` environment variable must contain the right target triplet corresponding to the AArch64 GCC compiler. Below is an example when RMM is to be built for AArch64 target on a non-native host machine and using GCC as the toolchain.

```
export CROSS_COMPILE=aarch64-none-elf-
export PATH=<path-to-aarch64-gcc>/bin:$PATH
```

Please note that AArch64 GCC must be included in the shell's search path even when using Clang as the compiler as LLVM does not include some C standard headers like *stdlib.h* and needs to be picked up from the *include* folder of the AArch64 GCC. Below is an example when RMM is to be built for AArch64 target on a non-native host machine and using LLVM as the toolchain.

```
export CROSS_COMPILE=aarch64-none-elf-
export PATH=<path-to-aarch64-gcc>/bin:<path-to-clang+llvm>/bin:$PATH
```

The `CROSS_COMPILE` variable is ignored for `fake_host` build and the native host toolchain is used for the build.

2.5 Package Installation (Ubuntu-20.04 x64)

If you are using the recommended Ubuntu distribution then we can install the required packages with the following commands:

1. Install dependencies:

```
sudo apt-get install -y git build-essential python3 python3-pip make ninja-build
sudo snap install cmake
```

2. Verify cmake version:

```
cmake --version
```

Note: Please download cmake 3.19 or later version from https://cmake.org/download/.

3. Add CMake path into environment:

```
export PATH=<CMake path>/bin:$PATH
```

2.6 Install python dependencies

Note: The installation of Python dependencies is an optional step. This is required only if building documentation.

RMM's `docs/requirements.txt` file declares additional Python dependencies. Install them with `pip3`:

```
pip3 install --upgrade pip
cd <rmm source folder>
pip3 install -r docs/requirements.txt
```

2.7 Install coverage tools analysis dependencies

Note: This is an optional step only needed if you intend to run coverage analysis on the source code.

On Ubuntu, `gcovr` tool can be installed in two different ways:

Using the pagckage manager:

```
sudo apt-get install gcovr
```

The second (and recommended) way is install it with `pip3`:

```
pip3 install --upgrade pip
pip3 install gcovr
```

2.8 Getting the RMM Source

Source code for *RMM* is maintained in a Git repository hosted on TrustedFirmware.org. To clone this repository from the server, run the following in your shell:

```
git clone --recursive https://git.trustedfirmware.org/TF-RMM/tf-rmm.git
```

2.8.1 Additional steps for Contributors

If you are planning on contributing back to RMM, your commits need to include a `Change-Id` footer as explained in *Mandated Trailers*. This footer is generated by a Git hook that needs to be installed inside your cloned RMM source folder.

The TF-RMM Gerrit page under trustedfirmware.org contains a *Clone with commit-msg hook* subsection under its **Download** header where you can copy the command to clone the repo with the required git hooks. Please use the **SSH** option to clone the repository on your local machine.

If needed, you can also manually install the hooks separately on an existing repo:

```
curl -Lo $(git rev-parse --git-dir)/hooks/commit-msg https://review.trustedfirmware.org/
↪tools/hooks/commit-msg
chmod +x $(git rev-parse --git-dir)/hooks/commit-msg
```

You can read more about Git hooks in the *githooks* page of the Git hooks documentation.

2.9 Install Cppcheck and dependencies

Note: The installation of Cppcheck is an optional step. This is required only if using the Cppcheck static analysis.

Follow the public documentation to install Cppcheck either from the official website https://cppcheck.sourceforge.io/ or from the official github https://github.com/danmar/cppcheck/

If you own a valid copy of a MISRA rules file:

```
cp -a <path to the misra rules file>/<file name> ${RMM_SOURCE_DIR}/tools/cppcheck/misra.
↪rules
```

2.10 Install CBMC

Note: The installation of CBMC is an optional step. This is required only if running source code analysis with CBMC.

Follow the public documentation to install CBMC either from the official website https://www.cprover.org/cbmc/ or from the official github https://github.com/diffblue/cbmc

2.11 Performing an Initial Build

The *RMM* sources can be compiled using multiple CMake options.

For detailed instructions on build configurations and examples see *RMM Build Examples*.

A typical build command for the FVP platform using GCC toolchain is shown below:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR}
```

2.12 Running the RMM

The *RMM* is part of the CCA software stack and relies on EL3 Firmware to load the binary at boot time appropriately. It needs both EL3 Firmware and Non-Secure Host to be present at runtime for its functionality. The EL3 Firmware must comply to RMM-EL3 Communication Specification and is typically the TF-A. The Non-Secure Host can be an RME aware hypervisor or an appropriate Test utility running in Non-Secure world which can interact with *RMM* via Realm Management Interface (RMI).

The TF-A project includes build and run instructions for an RME enabled system on the FVP platform as part of TF-A RME documentation. The `rmm.img` binary is provided to the TF-A bootloader to be packaged in FIP using `RMM` build option in TF-A.

If *RMM* is built for the *fake_host* architecture (see *RMM Fake Host Build*), then the generated *rmm.elf* binary can run natively on the Host machine. It does this by emulating parts of the system as described in *RMM Fake host architecture* design.

2.13 RMM Build Examples

The *RMM* supports a wide range of build configuration options. Some of these options are more regularly exercised by developers, while others are for **advanced** and **experimental** usage only.

RMM can be built using either GNU(GCC) or *LLVM(Clang)* toolchain. See *this section* for toolchain setup and the supported versions.

The build is performed in 2 stages:

Configure Stage: In this stage, a default config file can be specified which configures a sane config for the chosen platform. If this default config needs to be modified, it is recommended to first perform a default config and then modify using the cmake ncurses as shown in *CMake UI Example*.

Build Stage: In this stage, the source build is performed by specifying the *–build* option. See any of the commands below for an example.

Note: It is recommended to clean build if any of the build options are changed from previous build.

Below are some of the typical build and configuration examples frequently used in *RMM* development for the FVP Platform. Detailed configuration options are described *here*.

RMM also supports a `fake_host` build which can be used to build RMM for test and code analysis on the host machine. See *this section here* for more details.

1. Perform an initial default build with minimum configuration options:

Build using gnu toolchain

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR}
```

Build using LLVM toolchain

```
cmake -DRMM_CONFIG=fvp_defcfg -DRMM_TOOLCHAIN=llvm -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_
↪DIR}
cmake --build ${RMM_BUILD_DIR}
```

2. Perform an initial default config, then modify using ccmake ncurses UI:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
ccmake -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR}
```

3. Perform a debug build and specify a log level:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR} -DCMAKE_BUILD_
↪TYPE=Debug -DLOG_LEVEL=50
cmake --build ${RMM_BUILD_DIR}
```

4. Perform a documentation build:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR} -DRMM_DOCS=ON
cmake --build ${RMM_BUILD_DIR} -- docs
```

5. Perform a clean verbose build:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} --clean-first --verbose
```

6. Perform a build with Ninja Genenerator:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR} -DCMAKE_BUILD_
↪TYPE=${BUILD_TYPE} -G "Ninja" -DLOG_LEVEL=50
cmake --build ${RMM_BUILD_DIR}
```

7. Perform a build with Ninja Multi Config Genenerator:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR} -G "Ninja Multi-
↪Config" -DLOG_LEVEL=50
cmake --build ${RMM_BUILD_DIR} --config ${BUILD_TYPE}
```

8. Perform a Cppcheck static analysis:

```
cmake -DRMM_CONFIG=fvp_defcfg -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -S ${RMM_SOURCE_DIR} -B
↪${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- cppcheck
cat ${BUILD_DIR}/tools/cppcheck/cppcheck.xml
```

9. Perform a Cppcheck static analysis with MISRA:

```
cmake -DRMM_CONFIG=fvp_defcfg -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -S ${RMM_SOURCE_DIR} -B
↪${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- cppcheck-misra
cat ${BUILD_DIR}/tools/cppcheck/cppcheck_misra.xml
```

10. Perform a checkpatch analysis:

Run checkpatch on commits in the current branch against BASE_COMMIT (default origin/master):

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkpatch
```

Run checkpatch on entire codebase:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkcodebase
```

11. Perform a checkspdx analysis:

Run checkspdx on commits in the current branch against BASE_COMMIT (default origin/master):

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkspdx-patch
```

Run checkspdx on entire codebase:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkspdx-codebase
```

13. Check header file include order:

Run checkincludes-patch on commits in the current branch against BASE_COMMIT (default origin/master):

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkincludes-patch
```

Run checkincludes on entire codebase:

```
cmake -DRMM_CONFIG=fvp_defcfg -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- checkincludes-codebase
```

14. Perform a clang-tidy analysis:

Run clang-tidy on commits in the current branch against BASE_COMMIT (default origin/master):

```
cmake -DRMM_CONFIG=fvp_defcfg -DRMM_TOOLCHAIN=llvm -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -S
↪${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- clang-tidy-patch
```

Run clang-tidy on entire codebase:

```
cmake -DRMM_CONFIG=fvp_defcfg -DRMM_TOOLCHAIN=llvm -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -S
↪${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- clang-tidy-codebase
```

Note that clang-tidy will work with all configurations. It will only check the source files that are used for the specified configuration.

15. Perform unit tests on development host:

Build and run unit tests on host platform. It is recommended to enable the Debug build of RMM.

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_test -DCMAKE_BUILD_TYPE=Debug -S $
↪{RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- run-unittests
```

Run unittests for a specific test group(s) (e.g. unittests whose group starts with 'xlat')

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_test -DCMAKE_BUILD_TYPE=Debug -S $
↪{RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- build -j
${RMM_BUILD_DIR}/Debug/rmm.elf -gxlat -v -r${NUMBER_OF_TEST_ITERATIONS}
```

16. Generate Coverage Report.

It is possible to generate a coverage report for a last execution of the host platform (whichever the variant) by using the *run-coverage* build target.

For example, to generate coverate report on the whole set of unittests:

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_test -DRMM_COVERAGE=ON -DCMAKE_BUILD_
↪TYPE=Debug -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- run-unittests
cmake --build ${RMM_BUILD_DIR} -- run-coverage
```

Run coverage analysis on a specific set of unittests (e.g. unittests whose group starts with 'xlat')

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_test -DRMM_COVERAGE=ON -DCMAKE_BUILD_
↪TYPE=Debug -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- build -j
${RMM_BUILD_DIR}/Debug/rmm.elf -gxlat
cmake --build ${RMM_BUILD_DIR} -- run-coverage
```

Run coverage analysis on the *host_build* variant of host platform:

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_build -DRMM_COVERAGE=ON -DCMAKE_BUILD_
↪TYPE=Debug -S ${RMM_SOURCE_DIR} -B ${RMM_BUILD_DIR}
${RMM_BUILD_DIR}/Debug/rmm.elf
cmake --build ${RMM_BUILD_DIR} -- run-coverage
```

The above commands will automatically generate the HTML coverage report in folder *build/Debug/coverage* within the build directory. The HTML generation can be disabled by setting *RMM_HTML_COV_REPORT=OFF*.

17. Run CBMC analysis:

Run `COVERAGE`, `ANALYSIS` and `ASSERT` targets for CBMC. The results are generated in `${RMM_BUILD_DIR}/tools/cbmc/cbmc_coverage_results`.

```
cmake -DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_cbmc -S ${RMM_SOURCE_DIR} -B ${RMM_
↪BUILD_DIR}
cmake --build ${RMM_BUILD_DIR} -- cbmc-coverage cbmc-analysis cbmc-assert
```

2.14 RMM Build Options

The *RMM* build system supports the following CMake build options.

Table 2: RMM CMake Options Table

Option	Valid values	Default	Description
RMM_CONFIG			Platform build configuration, eg: fvp_defcfg for the FVP
RMM_ARCH	aarch64 | fake_host	aarch64	Target Architecture for RMM build
RMM_MAX_SIZE		0x0	Maximum size for RMM image
MAX_CPUS		16	Maximum number of CPUs supported by RMM
GRANULE_SHIFT		12	Granule Shift used by RMM
RMM_CCA_TOKEN_BUFFER		1	Number of pages to allocate in Aux granules for Realm CCA token
RMM_DOCS	ON | OFF	OFF	RMM Documentation build
CMAKE_BUILD_TYPE	Debug | Release	Release	CMake Build type
CMAKE_CONFIGURATION_TYPES	Debug & Release	Debug & Release	Multi-generator configuration types
CMAKE_DEFAULT_BUILD_TYPE	Debug | Release	Release	Default multi-generator configuration type
MbedTLS_BUILD_TYPE	Debug | Release	Release	MbedTLS build type
RMM_PLATFORM	fvp | host		Platform to build
RMM_TOOLCHAIN	gnu | llvm		Toolchain name

LOG_LEVEL	0 - 50	40(Debug)
		20(Release)
			Log level to apply for RMM (0 - 50).

RMM_STATIC_ANALYSIS			Enable static analysis checkers
RMM_UART_ADDR		0x0	Base addr of UART to be used for RMM logs
PLAT_CMN_CTX_MAX_XLAT_TABLES		0	Maximum number of translation tables used by the runtime context
PLAT_CMN_EXTRA_MMAP_REGIONS		0	Extra platform mmap regions that need to be mapped in S1 xlat tables
RMM_NUM_PAGES_PER_STACK		5	Number of pages to use per CPU stack
MBEDTLS_ECP_MAX_OPS	248 -	1000	Number of max operations per ECC signing iteration

RMM_FPU_USE_AT_REL2	ON | OFF	OFF(fake_host)
		ON(aarch64)
			Enable FPU/SIMD usage in RMM.

RMM_MAX_GRANULES		0	Maximum number of memory granules available to the system
HOST_VARIANT	host_build | host_test | host_cbmc	host_build	Variant to build for the host platform. Only available when RMM_PLATFORM=host
HOST_MEM_SIZE		0x40000000	Host memory size that will be used as physical granules
RMM_COVERAGE	ON | OFF	OFF	Enable coverage analysis
RMM_HTML_COV_REPORT	ON | OFF	ON	Enable HTML output report for coverage analysis
RMM_CBMC_VIEWER_OUTPUT	ON | OFF	OFF	Generate report of CBMC results using the tool cbmc-viewer
RMM_CBMC_SINGLE_TESTBENCH		OFF	Run CBMC on a single testbench instead on all of them

2.15 RMM LLVM Build

RMM can be built using LLVM Toolchain (Clang). To build using LLVM toolchain, set RMM_TOOLCHAIN=llvm during configuration stage.

2.16 RMM Fake Host Build

RMM also provides a `fake_host` target architecture which allows the code to be built natively on the host using the host toolchain. To build for `fake_host` architecture, set RMM_CONFIG=host_defcfg during the configuration stage.

CHAPTER

THREE

PROCESS

3.1 Coding Standard

This document describes the coding rules to follow to contribute to the project.

3.1.1 General

The following coding standard is derived from MISRA C:2012 Guidelines, TF-A coding style and Linux kernel coding style coding standards.

3.1.2 File Encoding

The source code must use the **UTF-8** character encoding. Comments and documentation may use non-ASCII characters when required (e.g. Greek letters used for units) but code itself is still limited to ASCII characters.

3.1.3 Language

The primary language for comments and naming must be International English. In cases where there is a conflict between the American English and British English spellings of a word, the American English spelling is used.

Exceptions are made when referring directly to something that does not use international style, such as the name of a company. In these cases the existing name should be used as-is.

3.1.4 C Language Standard

The C language mode used for *RMM* is *GNU11*. This is the “GNU dialect of ISO C11”, which implies the *ISO C11* standard with GNU extensions.

Both GCC and Clang compilers have support for *GNU11* mode, though Clang does lack support for a small number of GNU extensions. These missing extensions are rarely used, however, and should not pose a problem.

3.1.5 Length

- Each file, function and scopes should have a logical uniting theme.

  No length limit is set for a file.

- A function should be 24 lines maximum.

  This will not be enforced, any function being longer should trigger a discussion during the review process.

- The recommended maximum line length is 80 characters, except for string literals as it would make any search for it more difficult. A maximum length of 100 characters is enforced by the coding guidelines static check.

- A variable should not be longer than 31 characters.

  Although the C11 specification specifies that the number of signitificant characters in an identifier is implementation defined it sets the translation limit to the 31 initial characters.

TYPE	LIMIT
function	24 lines (not enforced)
line	100 characters
identifier	31 characters

3.1.6 Headers/Footers

- Include guards:

```
#ifndef FILE_NAME_H
#define FILE_NAME_H

<header content>

#endif /* FILE_NAME_H */
```

- Include statement variant is <>:

```
#include <file.h>
```

- Include files should be alphabetically ordered:

```
#include <axxxx.h>
#include <bxxxx.h>
[...]
#include <zxxxx.h>
```

- If possible, use forward declaration of struct types in public headers. This will reduce interdependence of header file inclusion.

```
#include <axxxx.h>
#include <bxxxx.h>
[...]
/* forward declaration */
struct x;
void foo(struct *x);
```

3.1.7 Naming conventions

- Case: Functions and variables must be in Snake Case

```c
unsigned int my_snake_case_variable = 0U;

void my_snake_case_function(void)
{
        [...]
}
```

- Local variables should be declared at the top of the closest opening scope and should be short.

  We won't enforce a length, and defining short is difficult, this motto (from Linux) catches the spirit

> LOCAL variable names should be short, and to the point.
> If you have some random integer loop counter, it should probably be called i.
> Calling it loop_counter is non-productive, if there is no chance of it being mis-understood.
> Similarly, tmp can be just about any type of variable that is used to hold a temporary value.
> If you are afraid to mix up your local variable names, you have another problem.

```c
int foo(const int a)
{
        int c; /* needed in the function */
        c = a; /* MISRA-C rules recommend to not modify arguments variables */

        if (c == 42) {
                int b; /* needed only in this "if" statment */

                b = bar(); /* bar will return an int */
                if (b != -1) {
                        c += b;
                }
        }
        return c;
}
```

- Use an appropraite prefix for public API of a component. For example, if the component name is *bar*, then the init API of the component should be called *bar_init()*.

3.1.8 Indentation

Use **tabs** for indentation. The use of spaces for indentation is forbidden except in the case where a term is being indented to a boundary that cannot be achieved using tabs alone.

Tab spacing should be set to **8 characters**.

Trailing whitespaces or tabulations are not allowed and must be trimmed.

3.1.9 Spacing

Single spacing should be used around most operators, including:

- Arithmetic operators (+, -, /, *, %)
- Assignment operators (=, +=, etc)
- Boolean operators (&&, ||)
- Comparison operators (<, >, ==, etc)
- Shift operators (>>, <<)
- Logical operators (&, |, etc)
- Flow control (`if`, `else`, `switch`, `while`, `return`, etc)

No spacing should be used around the following operators

- Cast (())
- Indirection (*)

3.1.10 Braces

- Use K&R style for statements.
- Function opening braces are on a new line.
- Use braces even for singled line.

```
void function(void)
{
        /* if statement */
        if (my_test) {
                do_this();
                do_that();
        }

        /* if/else statement */
        if (my_Test) {
                do_this();
                do_that();
        } else {
                do_other_this();
        }
}
```

3.1.11 Commenting

Double-slash style of comments (//) is not allowed, below are examples of correct commenting.

```
/*
 * This example illustrates the first allowed style for multi-line comments.
 *
 * Blank lines within multi-lines are allowed when they add clarity or when
 * they separate multiple contexts.
 */
```

```
/**************************************************************************
 * This is the second allowed style for multi-line comments.
 *
 * In this style, the first and last lines use asterisks that run the full
 * width of the comment at its widest point.
 *
 * This style can be used for additional emphasis.
 *************************************************************************/
```

```
/* Single line comments can use this format */
```

```
/***************************************************************************
 * This alternative single-line comment style can also be used for emphasis.
 **************************************************************************/
```

3.1.12 Error return values and Exception handling

- Function return type must be explicitly defined.
- Unless specifed otherwise by an official specification, return values must be used to return success or failure (Standard Posix error codes).

  Return an integer if the function is an action or imperative command

  Failure: -Exxx (STD posix error codes, unless specified otherwise)

  Success: 0

  Return a boolean if the function is as predicate

  Failure: false

  Success: true

- If a function returns error information, then that error information shall be tested.

  Exceptions are allowed for STDLIB functions (memcpy/printf/. . .) in which case it must be void casted.

```
#define MY_TRANSFORMED_ERROR  (-1)

void my_print_function(struct my_struct in_mystruct)
{
        long long transformed_a = my_transform_a(in_mystruct.a);

        if (transform_a != MY_TRANSFORMED_ERROR) {
                (void)printf("STRUCT\n\tfield(a): %ll\n", transformed_a);
```

(continues on next page)

(continued from previous page)

```
        } else {
                (void)printf("STRUCT\n\tERROR %ll\n", transformed_a);
        }
}
```

3.1.13 Use of asserts and panic

Assertions, as a general rule, are only used to catch errors during development cycles and are removed from production binaries. They are useful to document pre-conditions for a function or impossible conditions in code. They are not substitutes for proper error checking and any expression used to test an assertion must not have a side-effect.

For example,

```
assert(--i == 0);
```

should not be used in code.

Assertions can be used to validate input arguments to an API as long as the caller and callee are within the same trust boundary.

`panic()` is used in places wherein it is not possible to continue the execution of program sensibly. It should be used sparingly within code and, if possible, instead of panic(), components should return error back to the caller and the caller can decide on the appropriate action. This is particularly useful to build resilence to the program wherein non-functional part of the program can be disabled and, if possible, other functional aspects of the program can be kept running.

3.1.14 Using COMPILER_ASSERT to check for compile time data errors

Where possible, use the `COMPILER_ASSERT` macro to check the validity of data known at compile time instead of checking validity at runtime, to avoid unnecessary runtime code.

For example, this can be used to check that the assembler's and compiler's views of the size of an array is the same.

```
#include <utils_def.h>

define MY_STRUCT_SIZE 8 /* Used by assembler source files */

struct my_struct {
    uint32_t arg1;
    uint32_t arg2;
};

COMPILER_ASSERT(MY_STRUCT_SIZE == sizeof(struct my_struct));
```

If `MY_STRUCT_SIZE` in the above example were wrong then the compiler would emit an error like this:

```
my_struct.h:10:1: note: in expansion of macro 'COMPILER_ASSERT'
 10 | COMPILER_ASSERT(MY_STRUCT_SIZE == sizeof(struct my_struct));
    | ^~~~~~~~~~~~~~~
```

3.1.15 Data types, structures and typedefs

- Data Types:

The *RMM* codebase should be kept as portable as possible for 64-bits platforms. To help with this, the following data type usage guidelines should be followed:

- Where possible, use the built-in *C* data types for variable storage (for example, `char`, `int`, `long long`, etc) instead of the standard *C11* types. Most code is typically only concerned with the minimum size of the data stored, which the built-in *C* types guarantee.
- Avoid using the exact-size standard *C11* types in general (for example, `uint16_t`, `uint32_t`, `uint64_t`, etc) since they can prevent the compiler from making optimizations. There are legitimate uses for them, for example to represent data of a known structure. When using them in a structure definition, consider how padding in the structure will work across architectures.
- Use `int` as the default integer type - it's likely to be the fastest on all systems. Also this can be assumed to be 32-bit as a consequence of the Procedure Call Standard for the Arm 64-bit Architecture .
- Avoid use of `short` as this may end up being slower than `int` in some systems. If a variable must be exactly 16-bit, use `int16_t` or `uint16_t`.
- `long` are defined as LP64 (64-bit), this is guaranteed to be 64-bit.
- Use `char` for storing text. Use `uint8_t` for storing other 8-bit data.
- Use `unsigned` for integers that can never be negative (counts, indices, sizes, etc). *RMM* intends to comply with MISRA "essential type" coding rules (10.X), where signed and unsigned types are considered different essential types. Choosing the correct type will aid this. MISRA static analysers will pick up any implicit signed/unsigned conversions that may lead to unexpected behaviour.
- For pointer types:
  - If an argument in a function declaration is pointing to a known type then simply use a pointer to that type (for example: `struct my_struct *`).
  - If a variable (including an argument in a function declaration) is pointing to a general, memory-mapped address, an array of pointers or another structure that is likely to require pointer arithmetic then use `uintptr_t`. This will reduce the amount of casting required in the code. Avoid using `unsigned long` or `unsigned long long` for this purpose; it may work but is less portable.
  - Use of `void *` is generally discouraged. Although it is useful to represent pointers to types that are abstracted away from the callers and has useful implicit cast properties, for the sake of a more uniform code base, we encourage use of `uintptr_t` where possible.
  - Avoid pointer arithmetic generally (as this violates MISRA C 2012 rule 18.4) and especially on void pointers (as this is only supported via language extensions and is considered non-standard). In *RMM*, setting the `W` build flag to `W=3` enables the *-Wpointer-arith* compiler flag and this will emit warnings where pointer arithmetic is used.
  - Use `ptrdiff_t` to compare the difference between 2 pointers.
- Use `size_t` when storing the `sizeof()` something.
- Use `ssize_t` when returning the `sizeof()` something from a function that can also return an error code; the signed type allows for a negative return code in case of error. This practice should be used sparingly.
- Use `uint64_t` to store the contents of an AArch64 register or represent a 64-bit value. Use of `unsigned long` or `u_register_t` for these purposes is discouraged.

These guidelines should be updated if additional types are needed.

- Typedefs:

Typedef should be avoided and used only to create opaque types. An opaque data type is one whose concrete data structure is not publicly defined. Opaque data types can be used on handles to resources that the caller is not expected to address directly.

```
/* File main.c */
#include <my_lib.h>

int main(void)
{
	context_t	*context;
	int		res;

	context = my_lib_init();

	res = my_lib_compute(context, "2x2");
	if (res == -EMYLIB_ERROR) {
		return -1
	}

	return res;
}
```

```
/* File my_lib.h */
#ifndef MY_LIB_H
#define MY_LIB_H

typedef struct my_lib_context {
  [...] /* whatever internal private variables you need in my_lib */
} context_t;

#endif /* MY_LIB_H */
```

3.1.16 Macros and Enums

- Favor functions over macros.
- Preprocessor macros and enums values are written in all uppercase text.
- A numerical value shall be typed.

```
/* Common C usage */
#define MY_MACRO 4UL

/* If used in C and ASM (included from a .S file) */
#define MY_MACRO UL(4)
```

- Expressions resulting from the expansion of macro parameters must be enclosed in parentheses.
- A macro parameter immediately following a # operator mustn't be immediately followed by a ## operator.

```
#define SINGLE_HASH_OP(x)             (#x)             /* allowed */
#define SINGLE_DOUBLE_HASH_OP(x, y)   (x ## y)         /* allowed */
#define MIXED_HASH_OP(x, y)           (#x ## y)        /* not allowed */
```

- Avoid defining macros that affect the control flow (i.e. avoid using return/goto in a macro).

- Macro with multiple statements can be enclosed in a do-while block or in a expression statement.

```
int foo(char **b);

#define M1(a, b)                      \
        do {                          \
              if ((a) == 5) {         \
                      foo((b));       \
              }                       \
        } while (false)

#define M2(a, b)              \
        ({                    \
        if ((a) == 5) {       \
                foo((b));     \
        }                     \
        })

int foo(char **b)
{
        return 42;
}

int main(int ac, char **av)
{
        if (ac == 1) {
                M1(ac, av);
        } else if (ac == 2) {
                M2(ac, av);
        } else {
                return -1;
        }

        return ac;
}
```

3.1.17 Switch statements

- Return in a *case* are allowed.
- Fallthrough are allowed as long as they are commented.
- Do not rely on type promotion between the switch type and the case type.

3.1.18 Inline assembly

- Favor C language over assembly language.
- Document all usage of assembly.
- Do not mix C and ASM in the same file.

3.1.19 Libc functions that are banned or to be used with caution

Below is a list of functions that present security risks.

libc function	Comments
`strcpy, wcscpy, strncpy`	use strlcpy instead
`strcat, wcscat, strncat`	use strlcat instead
`sprintf, vsprintf`	use snprintf, vsnprintf instead
`snprintf`	if used, ensure result fits in buffer i.e : snprintf(buf,size…) < size
`vsnprintf`	if used, inspect va_list match types specified in format string
`strtok, strtok_r, strsep`	Should not be used
`ato*`	Should not be used
`*toa`	Should not be used

The use of above functions are discouraged and will only be allowed in justified cases after a discussion has been held either on the mailing list or during patch review and it is agreed that no alternative to their use is available. The code containing the banned APIs must properly justify their usage in the comments.

The above restriction does not apply to Third Party IP code inside the `ext/` directory.

3.2 Security Handling

The generic security incident process can be found at TrustedFirmware.org security incident process.

3.3 Commit Style

When writing commit messages, please think carefully about the purpose and scope of the change you are making: describe briefly what the change does, and describe in detail why it does it. This helps to ensure that changes to the code-base are transparent and approachable to reviewers, and it allows us to keep a more accurate changelog. You may use Markdown in commit messages.

A good commit message provides all the background information needed for reviewers to understand the intent and rationale of the patch. This information is also useful for future reference. For example:

- What does the patch do?
- What motivated it?
- What impact does it have?
- How was it tested?
- Have alternatives been considered? Why did you choose this approach over another one?

- If it fixes an issue, include a reference.
  - Github prescribes a format for issue fixes that can be used within the commit message:

    ```
    Fixes TF-RMM/tf-rmm#<issue-number>
    ```

Commit messages are expected to be of the following form, based on conventional commits:

```
<type>[optional scope]: <description>

[optional body]

[optional trailer(s)]
```

The following *types* are permissible :

Type	Description
`feat`	A new feature
`fix`	A bug fix
`build`	Changes that affect the build system or external dependencies
`docs`	Documentation-only changes
`perf`	A code change that improves performance
`refactor`	A code change that neither fixes a bug nor adds a feature
`revert`	Changes that revert a previous change
`style`	Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc.)
`test`	Adding missing tests or correcting existing tests
`chore`	Any other change

The permissible *scopes* are more flexible, and we recommend that they match the directory where the patch applies (or where the main subject of the patch is, in case of changes accross several directories).

The following example commit message demonstrates the use of the `refactor` type and the `lib/arch` scope:

```
refactor(lib/arch): ...

This change introduces ....

Change-Id: ...
Signed-off-by: ...
```

In addition, the width of the commit message must be no more than 72 characters.

3.3.1 Mandated Trailers

Commits are expected to be signed off with the `Signed-off-by:` trailer using your real name and email address. You can do this automatically by committing with Git's `-s` flag.

There may be multiple `Signed-off-by:` lines depending on the history of the patch. See *License and Copyright for Contributions* for guidance on this.

Ensure that each commit also has a unique `Change-Id:` line. If you have cloned the repository using the "*Clone with commit-msg hook*" clone method, then this should be done automatically for you.

More details may be found in the Gerrit Change-Ids documentation.

3.4 Contributor's Guide

3.4.1 Getting Started

- Make sure you have a Github account and you are logged on review.trustedfirmware.org.
- Clone RMM on your own machine as described in *Getting the RMM Source*.
- If you plan to contribute a major piece of work, it is usually a good idea to start a discussion around it on the mailing list. This gives everyone visibility of what is coming up, you might learn that somebody else is already working on something similar or the community might be able to provide some early input to help shaping the design of the feature.
- If you intend to include Third Party IP in your contribution, please mention it explicitly in the email thread and ensure that the changes that include Third Party IP are made in a separate patch (or patch series).
- Create a local topic branch based on the RMM `main` branch.

3.4.2 Making Changes

- See the *License and Copyright for Contributions* section for guidance on license and copyright.
- Ensure commits adhere to the project's *Commit Style*.
- Make commits of logical units. See these general Git guidelines for contributing to a project.
- Keep the commits on topic. If you need to fix another bug or make another enhancement, please address it on a separate topic branch.
- Split the patch into manageable units. Small patches are usually easier to review so this will speed up the review process.
- Avoid long commit series. If you do have a long series, consider whether some commits should be squashed together or addressed in a separate topic.
- Follow the *Coding Standard*.
  - Use the static checks as shown in *RMM Build Examples* to perform checks like checkpatch, checkspdx, header files include order etc.
- Where appropriate, please update the documentation.
  - Consider whether the *Design* document or other in-source documentation needs updating.
- Ensure that each patch in the patch series compiles in all supported configurations. For generic changes, such as on the libraries, The *RMM Fake host architecture* should be able to, at least, build. Patches which do not compile will not be merged.
- Please test your changes and add suitable tests in the available test frameworks for any new functionality.
- Ensure that all CI automated tests pass. Failures should be fixed. They might block a patch, depending on how critical they are.

3.4.3 Submitting Changes

- Assuming the clone of the repo has been done as mentioned in the *Getting the RMM Source* and *origin* refers to the upstream repo, submit your changes for review targeting the `integration` branch. Create a topic that describes the target of your changes to help group related patches together.

  ```
  git push origin HEAD:refs/for/integration [-o topic=<your_topic>]
  ```

  Refer to the Gerrit Uploading Changes documentation for more details.

- Add reviewers for your patch:
  - At least one maintainer. See the list of *Maintainers*.
  - Alternatively, you might send an email to the TF-RMM mailing list to broadcast your review request to the community.
- The changes will then undergo further review by the designated people. Any review comments will be made directly on your patch. This may require you to do some rework. For controversial changes, the discussion might be moved to the TF-RMM mailing list to involve more of the community.
- The patch submission rules are the following. For a patch to be approved and merged in the tree, it must get a `Code-Review+2`.

  In addition to that, the patch must also get a `Verified+1`. This is usually set by the Continuous Integration (CI) bot when all automated tests passed on the patch. Sometimes, some of these automated tests may fail for reasons unrelated to the patch. In this case, the maintainers might (after analysis of the failures) override the CI bot score to certify that the patch has been correctly tested.

  In the event where the CI system lacks proper tests for a patch, the patch author or a reviewer might agree to perform additional manual tests in their review and the reviewer incorporates the review of the additional testing in the `Code-Review+1` to attest that the patch works as expected.

- When the changes are accepted, the *Maintainers* will integrate them.
  - Typically, the *Maintainers* will merge the changes into the `integration` branch.
  - If the changes are not based on a sufficiently-recent commit, or if they cannot be automatically rebased, then the *Maintainers* may rebase it on the `integration` branch or ask you to do so.
  - After final integration testing, the changes will make their way into the `main` branch. If a problem is found during integration, the *Maintainers* will request your help to solve the issue. They may revert your patches and ask you to resubmit a reworked version of them or they may ask you to provide a fix-up patch.

3.4.4 License and Copyright for Contributions

All new files should include the BSD-3-Clause SPDX license identifier where possible. When contributing code to us, the committer and all authors are required to make the submission under the terms of the *Developer Certificate of Origin*, confirming that the code submitted can (legally) become part of the project, and be subject to the same BSD-3-Clause license. This is done by including the standard Git `Signed-off-by:` line in every commit message. If more than one person contributed to the commit, they should also add their own `Signed-off-by:` line.

Files that entirely consist of contributions to this project should have a copyright notice and BSD-3-Clause SPDX license identifier of the form :

```
SPDX-License-Identifier: BSD-3-Clause
SPDX-FileCopyrightText: Copyright TF-RMM Contributors.
```

Patches that contain changes to imported Third Party IP files should retain their original copyright and license notices. If changes are made to the imported files, then add an additional `SPDX-FileCopyrightText` tag line as shown above.

CHAPTER

FOUR

DESIGN

4.1 RMM Locking Guidelines

This document outlines the locking requirements, discusses the implementation and provides guidelines for a deadlock free *RMM* implementation. Further, the document hitherto is based upon *RMM* Alpha-05 specification and is expected to change as the implementation proceeds.

4.1.1 Introduction

In order to meet the requirement for the *RMM* to be small, simple to reason about, and to co-exist with contemporary hypervisors which are already designed to manage system memory, the *RMM* does not include a memory allocator. It instead relies on an untrusted caller providing granules of memory used to hold both meta data to manage realms as well as code and data for realms.

To maintain confidentiality and integrity of these granules, the *RMM* implements memory access controls by maintaining awareness of the state of each granule (aka Granule State, ref *Implementation*) and enforcing rules on how memory granules can transition from one state to another and how a granule can be used depending on its state. For example, all granules that can be accessed by software outside the *PAR* of a realm are in a specific state, and a granule that holds meta data for a realm is in another specific state that prevents it from being used as data in a realm and accidentally corrupted by a realm, which could lead to internal failure in the *RMM*.

Due to this complex nature of the operations supported by the *RMM*, for example when managing page tables for realms, the *RMM* must be able to hold locks on multiple objects at the same time. It is a well known fact that holding multiple locks at the same time can easily lead to deadlocking the system, as for example illustrated by the dining philosophers problem [EWD310]. In traditional operating systems software such issues are avoided by defining a partial order on all system objects and always acquiring a lower-ordered object before a higher-ordered object. This solution was shown to be correct by Dijkstra [EWD625]. Solutions are typically obtained by assigning an arbitrary order based upon certain attributes of the objects, for example by using the memory address of the object.

Unfortunately, software such as the *RMM* cannot use these methods directly because the *RMM* receives an opaque pointer from the untrusted caller and it cannot know before locking the object if it is indeed of the expected state. Furthermore, MMU page tables are hierarchical data structures and operations on the page tables typically must be able to locate a leaf node in the hierarchy based on single value (a virtual address) and therefore must walk the page tables in their hierarchical order. This implies an order of objects in the same Granule State which is not known by a process executing in the *RMM* before holding at least one lock on object in the page table hierarchy. An obvious solution to these problems would be to use a single global lock for the *RMM*, but that would serialize all operations across all shared data structures in the system and severely impact performance.

4.1.2 Requirements

To address the synchronization needs of the *RMM* described above, we must employ locking and lock-free mechanisms which satisfies a number of properties. These are discussed below:

Critical Section

A critical section can be defined as a section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code [WS2001].

Further, access to shared resources without appropriate synchronization can lead to **race conditions**, which can be defined as a situation in which multiple threads or processes read and write a shared item and the final result depends on the relative timing of their execution [WS2001].

In terms of *RMM*, an access to a shared resource can be considered as a list of operations/instructions in program order that either reads from or writes to a shared memory location (e.g. the granule data structure or the memory granule described by the granule data structure, ref *Implementation*). It is also understood that this list of operations does not execute indefinitely, but eventually terminates.

We can now define our desired properties as follows:

Mutual Exclusion

Mutual exclusion can be defined as the requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources [WS2001].

The following example illustrates how an implementation might enforce mutual exclusion of critical sections using a lock on a valid granule data structure *struct granule *a*:

```
struct granule *a;
bool r;

r = try_lock(a);
if (!r) {
        return -ERROR;
}
critical_section(a);
unlock(a);
other_work();
```

We note that a process might fail to perform the *lock* operation on object *a* and return an error or successfully acquire the lock, execute the *critical_section()*, *unlock()* and then continue to make forward progress to *other_work()* function.

Deadlock Avoidance

A deadlock can be defined as a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something [WS2001].

In other words, one or more processes are trying to enter their critical sections but none of them make forward progress.

We can then define the deadlock avoidance property as the inverse scenario:

When one or more processes are trying to enter their critical sections, at least one of them makes forward progress.

A deadlock is a fatal event if it occurs in supervisory software such as the *RMM*. This must be avoided as it can render the system vulnerable to exploits and/or unresponsive which may lead to data loss, interrupted service and eventually economic loss.

Starvation Avoidance

Starvation can be defined as a situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen [WS2001].

Then starvation avoidance can be defined as, all processes that are trying to enter their critical sections eventually make forward progress.

Starvation must be avoided, because if one or more processes do not make forward progress, the PE on which the process runs will not perform useful work and will be lost to the user, resulting in similar issues like a deadlocked system.

Nested Critical Sections

A critical section for an object may be nested within the critical section for another object for the same process. In other words, a process may enter more than one critical section at the same time.

For example, if the *RMM* needs to copy data from one granule to another granule, and must be sure that both granules can only be modified by the process itself, it may be implemented in the following way:

```c
struct granule *a;
struct granule *b;
bool r;

r = try_lock(a);
if (!r) {
        return -ERROR;
}

/* critical section for granule a -- ENTER */

r = try_lock(b);
if (r) {
        /* critical section for granule b -- ENTER */
        b->foo = a->foo;
        /* critical section for granule b -- EXIT */
        unlock(b);
}

/* critical section for granule a -- EXIT */
unlock(a);
```

4.1.3 Implementation

The *RMM* maintains granule states by defining a data structure for each memory granule in the system. Conceptually, the data structure contains the following fields:

- Granule State
- Lock
- Reference Count

The Lock field provides mutual exclusion of processes executing in their critical sections which may access the shared granule data structure and the shared meta data which may be stored in the memory granule which is in one of the *RD*, *REC*, and Table states. Both the data structure describing the memory granule and the contents of the memory granule itself can be accessed by multiple PEs concurrently and we therefore require some concurrency protocol to avoid corruption of shared data structures. An alternative to using a lock providing mutual exclusion would be to design all operations that access shared data structures as lock-free algorithms, but due to the complexity of the data structures and the operation of the *RMM* we consider this too difficult to accomplish in practice.

The Reference Count field is used to keep track of references between granules. For example, an *RD* describes a realm, and a *REC* describes an execution context within that realm, and therefore an *RD* must always exist when a *REC* exists. To prevent the *RMM* from destroying an *RD* while a *REC* still exists, the *RMM* holds a reference count on the *RD* for each *REC* associated with the same realm, and only when the all the RECs in a realm have been destroyed and the reference count on an *RD* drops to zero, can the *RD* be destroyed and the granule be repurposed for other use.

Based on the above, we now describe the Granule State field and the current locking/refcount implementation:

- **UnDelegated:** These are granules for which *RMM* does not prevent the *PAS* of the granule from being changed by another agent to any value. In this state, the granule content access is not protected by granule::lock, as it is always subject to reads and writes from Non-Realm worlds.
- **Delegated:** These are granules with memory only accessible by the *RMM*. The granule content is protected by granule::lock. No reference counts are held on this granule state.
- **Realm Descriptor (RD):** These are granules containing meta data describing a realm, and only accessible by the *RMM*. Granule content access is protected by granule::lock. A reference count is also held on this granule for each associated *REC* granule.
- **Realm Execution Context (REC):** These are granules containing meta data describing a virtual PE running in a realm, and are only accessible by the *RMM*. The execution content access is not protected by granule::lock, because we cannot enter a realm while holding the lock. Further, the following rules apply with respect to the granule's reference counts:
  - A reference count is held on this granule when a *REC* is running.
  - As *REC* cannot be run on two PEs at the same time, the maximum value of the reference count is one.
  - When the *REC* is entered, the reference count is incremented (set to 1) atomically while granule::lock is held.
  - When the *REC* exits, the reference counter is released (set to 0) atomically with store-release semantics without granule::lock being held.
  - The *RMM* can access the granule's content on the entry and exit path from the *REC* while the reference is held.
- **Translation Table:** These are granules containing meta data describing virtual to physical address translation for the realm, accessible by the *RMM* and the hardware Memory Management Unit (MMU). Granule content access is protected by granule::lock, but hardware translation table walks may read the RTT at any point in time. Multiple granules in this state can only be locked at the same time if they are part of the same tree, and only in topological order from root to leaf. The topological order of concatenated root level RTTs is from the lowest

address to the highest address. The complete internal locking order for RTT granules is: RD -> [RTT] -> … -> RTT. A reference count is held on this granule for each entry in the RTT that refers to a granule:

  - Table s2tte.
  - Valid s2tte.
  - Valid_NS s2tte.
  - Assigned s2tte.

- **Data:** These are granules containing realm data, accessible by the *RMM* and by the realm to which it belongs. Granule content access is not protected by granule::lock, as it is always subject to reads and writes from within a realm. A granule in this state is always referenced from exactly one entry in an RTT granule which must be locked before locking this granule. Only a single DATA granule can be locked at a time on a given PE. The complete internal locking order for DATA granules is: RD -> RTT -> RTT -> … -> DATA. No reference counts are held on this granule type.

Locking

The *RMM* uses spinlocks along with the object state for locking implementation. The lock provides similar exclusive acquire semantics known from trivial spinlock implementations, however also allows verification of whether the locked object is of an expected state.

The data structure for the spinlock can be described in C as follows:

```
typedef struct {
        unsigned int val;
} spinlock_t;
```

This data structure can be embedded in any object that requires synchronization of access, such as the *struct granule* described above.

The following operations are defined on spinlocks:

Listing 1: **Typical spinlock operations**

```
/*
 * Locks a spinlock with acquire memory ordering semantics or goes into
 * a tight loop (spins) and repeatedly checks the lock variable
 * atomically until it becomes available.
 */
void spinlock_acquire(spinlock_t *l);

/*
 * Unlocks a spinlock with release memory ordering semantics. Must only
 * be called if the calling PE already holds the lock.
 */
void spinlock_release(spinlock_t *l);
```

The above functions should not be directly used for locking/unlocking granules, instead the following should be used:

Listing 2: **Granule locking operations**

```
/*
 * Acquires a lock (or spins until the lock is available), then checks
 * if the granule is in the `expected_state`. If the `expected_state`
```

(continues on next page)

(continued from previous page)

```
 * is matched, then returns `true`. Otherwise, releases the lock and
 * returns `false`.
 */
bool granule_lock_on_state_match(struct granule *g,
                                 enum granule_state expected_state);

/*
 * Used when we're certain of the state of an object (e.g. because we
 * hold a reference to it) or when locking objects whose reference is
 * obtained from another object, after that objects is locked.
 */
void granule_lock(struct granule *g,
                  enum granule_state expected_state);

/*
 * Obtains a pointer to a locked granule at `addr` if `addr` is a valid
 * granule physical address and the state of the granule at `addr` is
 * `expected_state`.
 */
struct granule *find_lock_granule(unsigned long addr,
                                  enum granule_state expected_state);

/* Find two granules and lock them in order of their address. */
return_code_t find_lock_two_granules(unsigned long addr1,
                                     enum granule_state expected_state1,
                                     struct granule **g1,
                                     unsigned long addr2,
                                     enum granule_state expected_state2,
                                     struct granule **g2);

/*
 * Obtain a pointer to a locked granule at `addr` which is unused
 * (refcount = 0), if `addr` is a valid granule physical address and the
 * state of the granule at `addr` is `expected_state`.
 */
struct granule *find_lock_unused_granule(unsigned long addr,
                                         enum granule_state
                                         expected_state);
```

Listing 3: **Granule unlocking operations**

```
/*
 * Release a spinlock held on a granule. Must only be called if the
 * calling PE already holds the lock.
 */
void granule_unlock(struct granule *g);

/*
 * Sets the state and releases a spinlock held on a granule. Must only
 * be called if the calling PE already holds the lock.
 */
void granule_unlock_transition(struct granule *g,
```

(continues on next page)

(continued from previous page)

```
                                    enum granule_state new_state);
```

Reference Counting

The reference count is implemented using the **refcount** variable within the granule structure to keep track of the references in between granules. For example, the refcount is used to prevent changes to the attributes of a parent granule which is referenced by child granules, ie. a parent with refcount not equal to zero.

Race conditions on the refcount variable are avoided by either locking the granule before accessing the variable or by lock-free mechanisms such as Single-Copy Atomic operations along with ARM weakly ordered ACQUIRE/RELEASE/RELAXED memory semantics to synchronize shared resources.

The following operations are defined on refcount:

Listing 4: **Read a refcount value**

```
/*
 * Single-copy atomic read of refcount variable with RELAXED memory
 * ordering semantics. Use this function if lock-free access to the
 * refcount is required with relaxed memory ordering constraints applied
 * at that point.
 */
unsigned long granule_refcount_read_relaxed(struct granule *g);

/*
 * Single-copy atomic read of refcount variable with ACQUIRE memory
 * ordering semantics. Use this function if lock-free access to the
 * refcount is required with acquire memory ordering constraints applied
 * at that point.
 */
unsigned long granule_refcount_read_acquire(struct granule *g);
```

Listing 5: **Increment a refcount value**

```
/*
 * Increments the granule refcount. Must be called with the granule
 * lock held.
 */
void __granule_get(struct granule *g);

/*
 * Increments the granule refcount by `val`. Must be called with the
 * granule lock held.
 */
void __granule_refcount_inc(struct granule *g, unsigned long val);

/* Atomically increments the reference counter of the granule.*/
void atomic_granule_get(struct granule *g);
```

Listing 6: **Decrement a refcount value**

```
/*
 * Decrements the granule refcount. Must be called with the granule
 * lock held.
 */
void __granule_put(struct granule *g);

/*
 * Decrements the granule refcount by `val`. Asserts if refcount can
 * become negative. Must be called with the granule lock held.
 */
void __granule_refcount_dec(struct granule *g, unsigned long val);

/* Atomically decrements the reference counter of the granule. */
void atomic_granule_put(struct granule *g);

/*
 * Atomically decrements the reference counter of the granule. Stores to
 * memory with RELEASE semantics.
 */
void atomic_granule_put_release(struct granule *g);
```

Listing 7: **Directly access refcount value**

```
/*
 * Directly reads/writes the refcount variable. Must be called with the
 * granule lock held.
 */
granule->refcount;
```

4.1.4 Guidelines

In order to meet the *Requirements* discussed above, this section stipulates some locking and lock-free algorithm implementation guidelines for developers.

Mutual Exclusion

The spinlock, acquire/release and atomic operations provide trivial mutual exclusion implementations for *RMM*. However, the following general guidelines should be taken into consideration:

- Appropriate deadlock avoidance techniques should be incorporated when using multiple locks.
- Lock-free access to shared resources should be atomic.
- Memory ordering constraints should be used prudently to avoid performance degradation. For e.g. on an unlocked granule (e.g. REC), prior to the refcount update, if there are associated memory operations, then the update should be done with release semantics. However, if there are no associated memory accesses to the granule prior to the refcount update then release semantics will not be required.

Deadlock Avoidance

Deadlock avoidance is provided by defining a partial order on all objects in the system where the locking operation will eventually fail if the caller tries to acquire a lock of a different state object than expected. This means that no two processes will be expected to acquire locks in a different order than the defined partial order, and we can rely on the same reasoning for deadlock avoidance as shown by Dijkstra [EWD625].

To establish this partial order, the objects referenced by *RMM* can be classified into two categories:

1. **External**: A granule state belongs to the *external* class iff _any_ parameter in _any_ RMI command is an address of a granule which is expected to be in that state. The following granule states are *external*:
   - GRANULE_STATE_NS
   - GRANULE_STATE_DELEGATED
   - GRANULE_STATE_RD
   - GRANULE_STATE_REC
2. **Internal**: A granule state belongs to the *internal* class iff it is not an *external*. These are objects which are referenced from another object after that object is locked. Each *internal* object should be referenced from exactly one place. The following granule states are *internal*:
   - GRANULE_STATE_RTT
   - GRANULE_STATE_DATA

We now state the locking guidelines for *RMM* as:

1. Granules expected to be in an *external* state must be locked before locking any granules in an *internal* state.
2. Granules expected to be in an *external* state must be locked in order of their physical address, starting with the lowest address.
3. Once a granule expected to be in an *external* state has been locked, its state must be checked against the expected state. If these do not match, the granule must be unlocked and no further granules may be locked within the currently-executing RMM command.
4. Granules in an *internal* state must be locked in order of state:
   - *RTT*
   - *DATA*
5. Granules in the same *internal* state must be locked in the *Implementation* defined order for that specific state.
6. A granule's state can be changed iff the granule is locked and the reference count is zero.

Starvation Avoidance

Currently, the lock-free implementation for RMI.REC.Enter provides Starvation Avoidance in *RMM*. However, for the locking implementation, Starvation Avoidance is yet to be accomplished. This can be added by a ticket or MCS style locking implementation [MCS].

Nested Critical Sections

Spinlocks provide support for nested critical sections. Processes can acquire multiple spinlocks at the same time, as long as the locking order is not violated.

4.1.5 References

4.2 MMU setup and memory management design in RMM

This document describes how the MMU is setup and how memory is managed by the *RMM* implementation.

4.2.1 Physical Address Space

The Realm Management Extension (`FEAT_RME`) defines four Physical Address Spaces (PAS):

- Non-secure
- Secure
- Realm
- Root

RMM code and *RMM* data are in Realm PAS memory, loaded and allocated to Realm PAS at boot time by the EL3 Firmware. This is a static carveout and it is never changed during the lifetime of the system.

The size of the *RMM* data is fixed at build time. The majority of this is the granules array (see *Granule state tracking* below), whose size is configurable and proportional to the maximum amount of delegable DRAM supported by the system.

Realm data and metadata are in Realm PAS memory, which is delegated to the Realm PAS by the Host at runtime. The *RMM* ABI ensures that this memory cannot be returned to Non-secure PAS ("undelegated") while it is in use by the *RMM* or by a Realm.

NS data is in Non-secure PAS memory. The Host is able to change the PAS of this memory while it is being accessed by the *RMM*. Consequently, the *RMM* must be able to handle a Granule Protection Fault (GPF) while accessing NS data as part of RMI handling.

4.2.2 Granule state tracking

The *RMM* manages a data structure called the *granules* array, which is stored in *RMM* data memory.

The *granules* array contains one entry for every Granule of physical memory which was in Non-secure PAS at *RMM* boot and can be delegated.

Each entry in the *granules* array contains a field *granule_state* which records the *state* of the Granule and which can be one of the states as listed below:

- NS: Not Realm PAS (i.e. Non-secure PAS, Root PAS or Secure PAS)
- Delegated: Realm PAS, but not yet assigned a purpose as either Realm data or Realm metadata
- RD: Realm Descriptor
- REC: Realm Execution Context
- REC aux: Auxiliary storage for REC

- Data: Realm data
- RTT: Realm Stage 2 translation tables

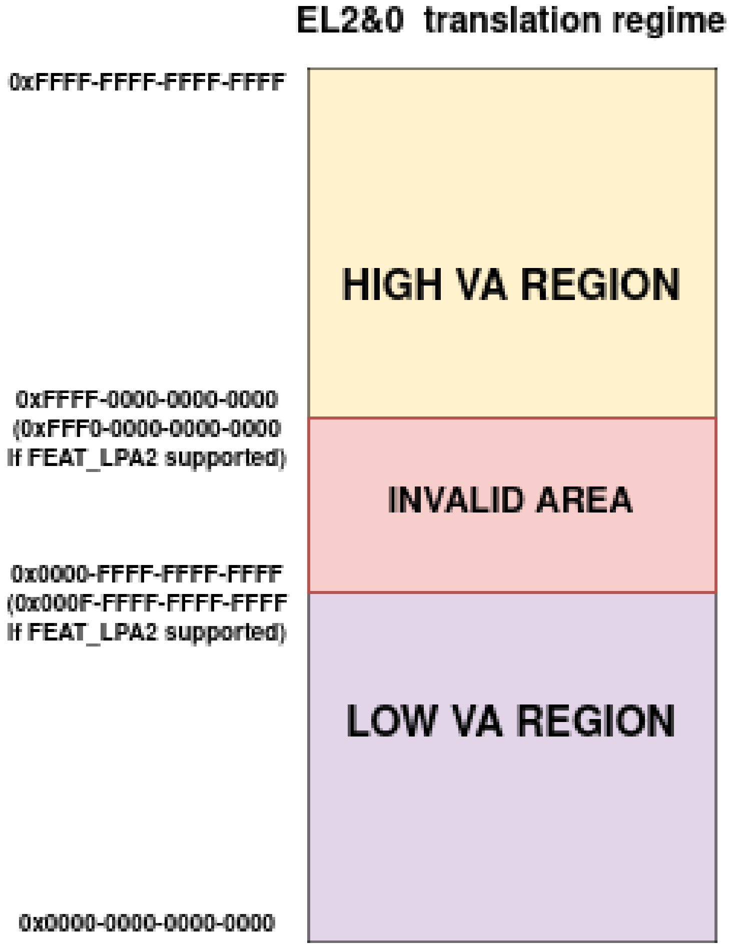
As part of RMI SMC handling, the state of the granule can be a pre-condition and undergo transition to a new state. For more details on the various granule states and their transitions, please refer to the Realm Management Monitor (RMM) Specification.

For further details, see:

- `enum granule_state`
- `struct granule`

4.2.3 RMM stage 1 translation regime

RMM uses the `FEAT_VHE` extension to split the 64-bit VA space into two address spaces as shown in the figure below:



- The Low VA range: it expands from VA 0x0 up to the maximum VA size configured for the region (with a maximum VA size of 48 bits or 52 bits if `FEAT_LPA2` is supported). This range is used to map the *RMM* Runtime (code, data, shared memory with EL3-FW and any other platform mappings).
- The High VA range: It expands from VA 0xFFFF_FFFF_FFFF_FFFF all the way down to an address corresponding to the maximum VA size configured for the region. This region is used by the *Stage 1 High VA - Slot*

Buffer mechanism as well as the *Per-CPU stack mapping*.

There is a range of invalid addresses between both ranges that is not mapped to any of them as shown in the figure above. TCR_EL2.TxSZ fields controls the maximum VA size of each region and *RMM* configures this field to fit the mappings used for each region.

The 2 VA ranges are used for 2 different purposes in RMM as described below.

Stage 1 Low VA range

The Low VA range is used to create static mappings which are shared across all the CPUs. It encompasses the RMM executable binary memory and the EL3 Shared memory region.

The RMM Executable binary memory consists of code, RO data and RW data. Note that the stage 1 translation tables for the Low Region are kept in RO data, so that once the MMU is enabled, the tables mappings are protected from further modification.

The EL3 shared memory, which is allocated by the EL3 Firmware, is used by the RMM-EL3 communications interface. A pointer to the beginning of this area is received by *RMM* during initialization. *RMM* will then map the region in the .rw area.

The Low VA range is setup by the platform layer as part of platform initialization.

The following mappings belong to the Low VA Range:

- RMM_CODE
- RMM_RO
- RMM_RW
- RMM_SHARED

Per-platform mappings can also be added if needed, such as the UART for the FVP platform.

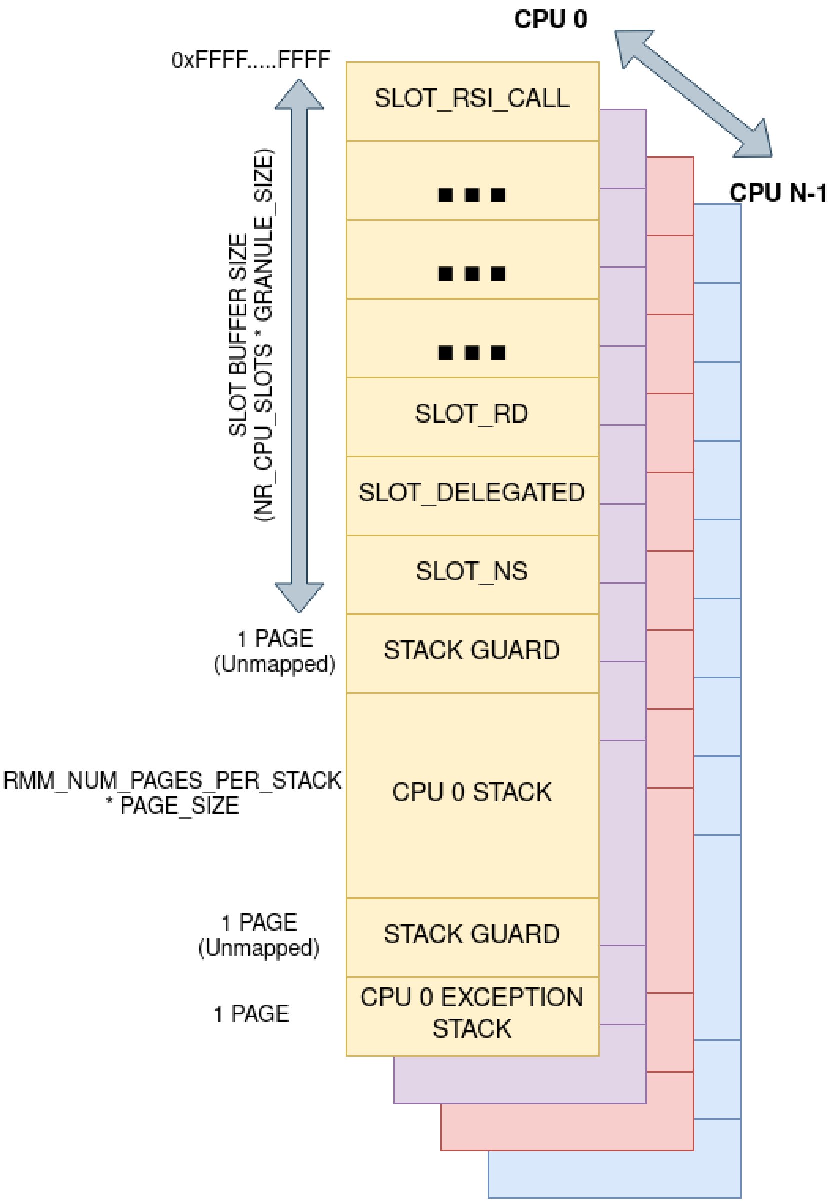
Stage 1 High VA range

The High VA range is used to create dynamic per-CPU mappings. The tables used for this are private to each CPU and hence it is possible for every CPU to map a different PA at a specific VA. This property is used by the *slot-buffer* mechanism as described later.

In order to allow the mappings for this region to be dynamic, its translation tables are stored in the RW section of *RMM*, allowing for it to be modified as needed.

For more details see `xlat_high_va.c` file of the xlat library.

The diagram below shows the memory layout for the High VA region.



Stage 1 High VA - Slot Buffer mechanism

The *RMM* provides a dynamic mapping mechanism called *slot-buffer* in the high VA region. The assigned VA space for *slot-buffer* is divided into *slots* of GRANULE_SIZE each.

The *RMM* has a fixed number of *slots* per CPU. Each *slot* is used to map memory of a particular category. The *RMM* validates that the target physical granule to be mapped is of the expected *granule_state* by looking up the corresponding entry in *granules* array.

The *slot-buffer* mechanism has *slots* for mapping memory of the following types:

- Realm metadata: These correspond to the specific Realm and Realm Execution context scheduled on the PE. These mappings are usually only valid during the execution of an RMI or RSI handlers and are removed afterwards. These include Realm Descriptors (RDs), Realm Execution Contexts (RECs), Realm Translation Tables (RTTs).

- NS data: RMM needs to map NS memory as part of RMIs to access parameters passed by the Host or to return arguments to the Host. RMM also needs to copy Data provided by the Host as part of populating the Realm data memory.
- Realm data: RMM sometimes needs to temporarily map Realm data memory during Realm creation in order to load the Realm image or access buffers specified by the Realm as part of RSI commends.

The *slot-buffer* design avoids the need for generic allocation of VA space. The rationalization of all mappings ever needed for managing a realm via *slots* is only possible due to the simple nature of the *RMM* design - in particular, the fact that it is possible to statically determine the types of objects which need to be mapped into the *RMM*'s address space, and the maximum number of objects of a given type which need to be mapped at any point in time.

During Realm entry and Realm exit, the RD is mapped in the "RD" buffer slot. Once Realm entry or Realm exit is complete, this mapping is removed. The RD is not mapped during Realm execution.

The REC and the *rmi_rec_run* data structures are both mapped during Realm execution.

As the *slots* are mapped on the High VA region, each CPU has its own private translation tables for such mappings, which means that a particular slot has a fixed VA on every CPU. Since the Translation tables are private to a CPU, the mapping to the slot is private to the CPU. This allows the interruption and migration of a REC (vCPU) to another CPU with live memory allocations in RMM. An example of this scenario is when the Realm attestation token is being created in RMM, a pending IRQ can cause RMM to yield to NS Host with live memory allocations in MbedTLS heap. The NS Host can schedule the REC on another CPU and, since the mapping for the memory allocations remain at the same VA, the interrupted realm token creation can continue.

The *slot-buffer* implementation in RMM also has some performance optimizations like caching of TTE's to avoid walking the Stage 1 translation tables for every map and unmap operation.

As an alternative to using dynamic mappings as required for the RMI command, the approach of maintaining static mappings for all physical memory was considered, but rejected on the grounds that this could permit arbitrary memory access for an attacker who is able to subvert *RMM* execution.

The xlat lib APIs are used by the *slot-buffer* to create dynamic mappings. These dynamic mappings are stored in the high VA region's `xlat_ctx` structure and marked by the xlat library as *TRANSIENT*. This helps xlat lib to distinguish valid Translation Table Entries from invalid ones as otherwise the unmapped dynamic TTEs would be identical to INVALID ones.

For further details, see:

- `enum buffer_slot`
- `lib/realm/src/buffer.c`

Per-CPU stack mapping

Each CPU maps its stack to the High VA region which means that the stack has same VA on all the CPUs and it is private to the CPU. At boot time, each CPU calculates the PA for the start of the stack and maps it to the designated High VA address space.

The per-CPU VA mapping also includes a gap at the end of the stack VA to detect any stack underflows. The gap has a page size.

RMM also uses a separate Per-CPU stack to handle exceptions and faults. This stack is allocated below the general one, and it allows for *RMM* to be able to handle a stack overflow fault. There is another page gap of unmapped memory between both stacks to harden security.

The rest of the VA space available below the exception stack is unused and therefore left unmapped. The stage 1 translation library will not allow to map anything there.

4.2.4 Stage 1 translation library (xlat library)

The *RMM* stage 1 translation management is taken care of by the xlat library. This library is able to support up to 52-bit addresses and 5 levels of translation (when `FEAT_LPA2` is enabled).

The xlat library is designed to be stateless and it uses the abstraction of *translation context*, modelled through the `struct xlat_ctx`. A translation context stores all the information related to a given VA space, such as the translation tables, the VA configuration used to initialize the context and any internal status related to such VA. Once a context has been initialized, its VA configuration cannot be modified.

At the moment, although the xlat library supports creation of multiple contexts, it assumes that the caller will only use a single context per CPU for a given VA region. The library does not offer support to switch contexts on a CPU at run time. A context can be shared by several CPUs if they share the same VA configuration and mappings, like on the low va region.

Dynamic mappings can be created by specifying the `TRANSIENT` flag. The high VA region create dynamic mappings using this flag.

For further details, see `lib/xlat`.

4.2.5 RMM executable bootstrap

The *RMM* is loaded as a .bin file by the EL3 loader. The size of the sections in the *RMM* binary as well as the placing of *RMM* code and data into appropriate sections is controlled by the linker script in the source tree.

Platform initialization code takes care of importing the linker symbols that define the boundaries of the different sections and creates static memory mappings that are then used to initialize an `xlat_ctx` structure for the low VA region. The RMM binary sections are flat-mapped and are shared across all the CPUs on the system. In addition, as *RMM* is compiled as a Position Independent Executable (PIE) at address 0x0, the Global Offset Table (GOT) and other relocations in the binary are fixed up with the right offsets as part of boot. This allows RMM to be run at any physical address as a PIE regardless of the compile time address.

For further details, see:

- `runtime/linker.lds`
- `plat/common/src/plat_common_init.c`
- `plat/fvp/src/fvp_setup.c`

4.3 RMM Folder and Component organization

4.3.1 Root Level Folders and Components

The root level folder structure of the RMM project is as given below.

```
── cmake
├── configs
├── docs
├── drivers
├── ext
├── lib
├── plat
```

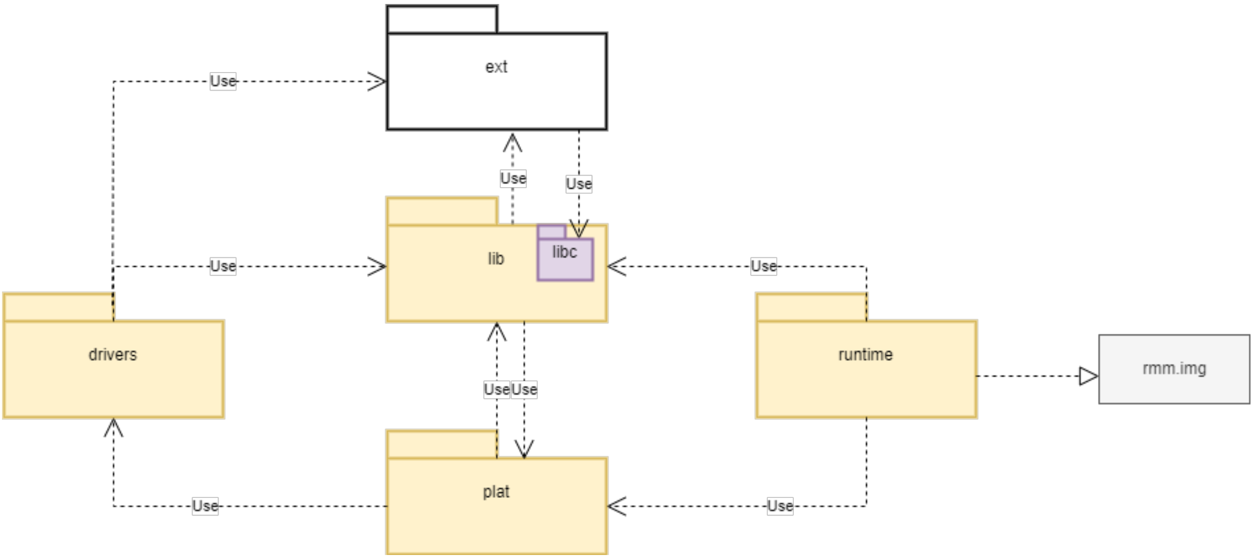
(continues on next page)

(continued from previous page)

```
├── runtime
├── toolchains
└── tools
```

The RMM functionality is implemented by files in lib, ext, drivers, plat and runtime. Each of these folders corresponds to a component in the project. Every component has a defined role in implementing the RMM functionality and can in-turn be composed of sub-components of the same role. The components have their own CMakelists.txt file and a defined public API which is exported via the public interface of the component to its dependent users. The runtime component is an exception as it does not have a public API.

The dependency relationship between the top level components is shown below :



Each component and its role is described below :

- **lib** : This component is a library of re-usable and architectural code which needs to be used by other components. The lib component is composed of several sub-components and every sub-component has a public API which is exported via its public interface. The functionality implemented by the sub-component is not platform specific although there could be specific static configuration or platform specific data provided via defined public interface. All of the sub-components in lib are combined into a single archive file which is then included in the build.

  The lib component depends on ext and plat components. All other components in the project depend on lib.

- **ext** : This component is meant for external source dependencies of the project. The sub folders are external open source projects configured as git submodules. The ext component is only allowed to depend on libc implementation in lib component.

- **plat** : This component implements the platform abstraction layer or platform layer for short. The platform layer has the following responsibilities:

  1. Implement the platform porting API as defined in platform_api.h.
  2. Do any necessary platform specific initialization in the platform layer.
  3. Initialize lib sub-components with platform specific data.
  4. Include any platform specific drivers from the drivers folder and initialize them as necessary.

  Every platform or a family of related platforms is expected to have a folder in plat and only one such folder corresponding to the platform will be included in the build. The plat component depends on lib and any platform specific drivers in drivers.

- **drivers** : The platform specific drivers are implemented in this component. Only the *plat* component is allowed to access these drivers via its public interface.
- **runtime** : This component implements generic RMM functionality which does not need to be shared across different components. The runtime component does not have a public interface and is not a dependency for any other component. The runtime is compiled into the binary `rmm.img` after linking with other components in the build.

4.3.2 Component File and Cmake Structure

The below figure shows the folder organization of a typical component (or sub-component)

```
component x
├── include
|   └── public.h
├── src
|   ├── private_a.h
|   └── src_a.c
├── tests
|   └── test.cpp
└── CMakeLists.txt
```

The `include` folder contains the headers exposing the public API of the component. The `src` contains the private headers and implementation of the intended functionality. The `tests` contains the tests for the component and the `CMakeLists.txt` defines the build and inheritance rules.

A typical component `CMakeLists.txt` has the following structure :

```
add_library(comp-x)

# Define any static config option for this component.
arm_config_option()

# Pass the config option to the source files as a compile
# option.
target_compile_definitions()

# Specify any private dependencies of the component. These are not
# inherited by child dependencies.
target_link_libraries(comp-x
    PRIVATE xxx)

# Specify any private dependencies of the component. These are
# inherited by child dependencies and are usually included in
# public API header of the component.
target_link_libraries(comp-x
    PUBLIC yyy)

# Export public API via public interface of this component
target_include_directories(comp-x
    PUBLIC "include")
```

(continues on next page)

(continued from previous page)

```
# Specify any private headers to be included for compilation
# of this component.
target_include_directories(comp-x
    PRIVATE "src")

# Specify source files for component
target_sources(comp-x
    PRIVATE xxx)
```

4.4 RMM Fake host architecture

RMM supports building and running the program natively as a regular user-space application on the host machine. It achieves this by emulating the `aarch64` specific parts of the program on the host machine by suitable hooks in the program. The implementation of the hooks can differ based on the target employment of running the program in this mode. Some of the foreseen employment scenarios of this architecture includes:

1. Facilitate development of architecture independent parts of RMM on the host machine.
2. Enable unit testing of components within RMM with the benefit of not having to mock all the dependencies of the component.
3. Leverage host development environment and tools for various purposes like debugging, measure code coverage, fuzz testing, stress testing, runtime analysis of program etc.
4. Enable RMM compliance testing and verification of state machine and locking rules on the host machine.
5. Profile RMM on the host machine and generate useful insights for possible optimizations.

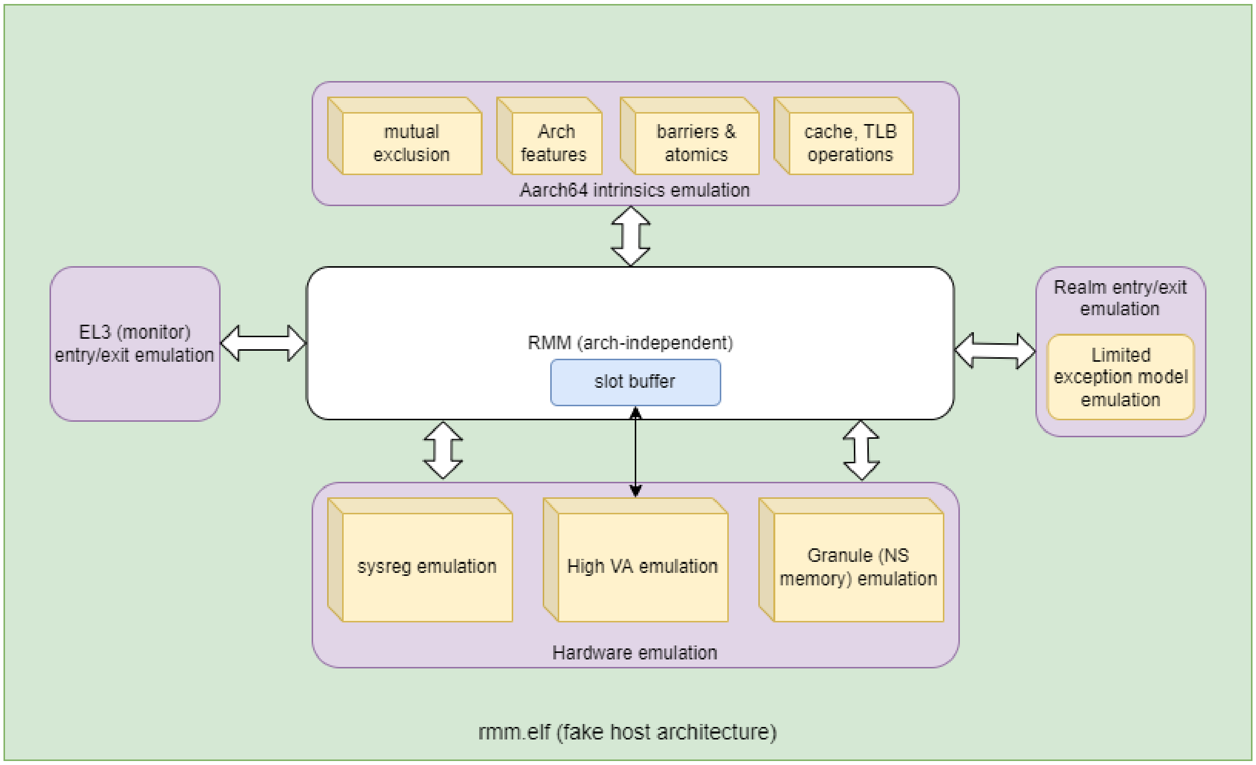
We expect the fake host architecture to be developed over time in future to cover some of the employment scenarios described above. The current code may not reflect the full scope of this architecture as discussed in this document.

The fake host architecture has some limitations:

1. The architecture is not intended to support multi-thread execution. The intrisics to support critical section and atomics are emulated as NOP.
2. Cannot execute AArch64 assembly code on the host due to obvious reasons.
3. Cannot emulate AArch64 exceptions during RMM execution although some limited form of handling exceptions occurring in Realms can probably be emulated.
4. The program links against the native compiler libraries which enables use of development and debug features available on the host machine. This means the libc implementation in RMM cannot be verified using this architecture.

The fake host architecture config is selected by setting the config `RMM_ARCH=fake_host` and the platform has to be set to a variant of *host* when building RMM. The different variants of the *host* platform allow to build RMM for each of the target employment scenarios as listed above.

4.4.1 Fake host architecture design


mutual exclusion
Arch features
barriers & atomics
cache, TLB operations
Aarch64 intrinsics emulation
EL3 (monitor) entry/exit emulation
RMM (arch-independent)
slot buffer
Realm entry/exit emulation
Limited exception model emulation
sysreg emulation
High VA emulation
Granule (NS memory) emulation
Hardware emulation
rmm.elf (fake host architecture)

The above figure shows the fake host architecture design. The architecture independent parts of RMM are linked against suitable host emulation blocks to enable the program to run on the host platform.

The EL3 (monitor) emulation layer emulates the entry and exception from EL3 into Realm-EL2. This includes entry and exit from RMM as part of RMI handling, entry into RMM as part of warm/cold boot, and EL3 service invocations by RMM using SMC calls. Similarly the Realm entry/exit emulation block allows emulation of running a Realm. It would also allow to emulate exit from Realm due to synchronous or asynchronous exceptions like SMC calls, IRQs, etc.

The hardware emulation block allows to emulate sysreg accesses, granule memory delegation and NS memory accesses needed for RMM. Since RMM is running as a user space application, it does not have the ability to map granule memory to a Virtual Address space. This capability is needed for the `slot buffer` component in RMM. Hence there is also need to emulate VA mapping for this case.

The AArch64 intrinsics emulation block allows emulation of exclusives, assembly instructions for various architecture extensions, barriers and atomics, cache and TLB operations although most of them are defined as NOP at the moment.

Within the RMM source tree, all files within the `fake_host` folder of each component implement the necessary emulation on host. Depending on the target employment for the fake host architecture, it is necessary to adapt the behaviour of the emulation layer. This is facilitated by the APIs defined in `host_harness.h` header. The implementation of the API is done by the `host` platform and each variant of the `host` can have a different implementation of the API suiting its target employment. The API also facilitates test and verification of the emulated property as needed by the employment.

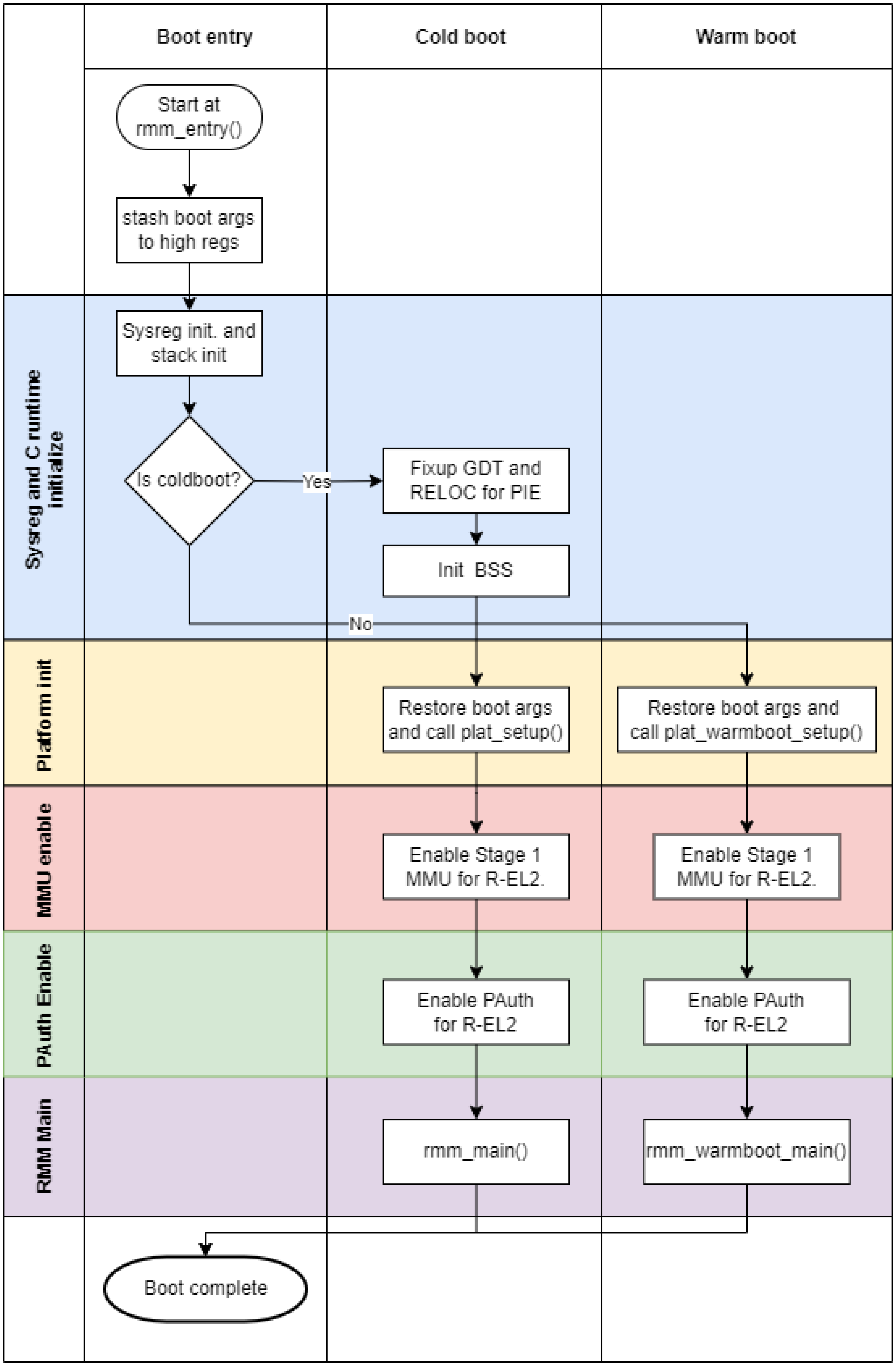
4.4.2 Fake host architecture employment scenarios implemented or ongoing

This section describes the currently implemented scenarios utilizing the fake host architecture.

1. Unit testing framework in RMM which allows testing public API of components and generation of code coverage data.

4.5 RMM Cold and Warm boot design

This section covers the boot design of RMM. The below diagram gives an overview of the boot flow.


Boot entry
Cold boot
Warm boot
Start at rmm_entry()
stash boot args to high regs
Sysreg and C runtime initialize
Sysreg init. and stack init
Is coldboot?
Yes
Fixup GDT and RELOC for PIE
Init BSS
No
Platform init
Restore boot args and call plat_setup()
Restore boot args and call plat_warmboot_setup()
MMU enable
Enable Stage 1 MMU for R-EL2.
Enable Stage 1 MMU for R-EL2.
PAuth Enable
Enable PAuth for R-EL2
Enable PAuth for R-EL2
RMM Main
rmm_main()
rmm_warmboot_main()
Boot complete

Both warm and cold boot enters RMM at the same entry point `rmm_entry()`. This scheme simplifies the RMM-EL3 communications interface. The boot args as specified by boot contract are stashed to high registers.

The boot is divided into several phases as described below:

1. **Sysreg and C runtime initialization phase.**

   The essential system registers are initialized. `SCTLR_EL2.I` is set to 1 which means instruction accesses to Normal memory are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable. `SCTLR_EL2.C` is also set 1 and data accesses default to Device-nGnRnE. The cpu-id, received as part of boot args, is programmed to `tpidr_el2` and this can be retrieved using the helper function `my_cpuid()`. The per-CPU stack is also initialized using the cpu-id received and this completes the C runtime initialization for warm boot.

   Only the primary CPU enters RMM during cold boot and a global variable is used to keep track whether it is cold or warm boot. If cold boot, the Global Descriptor Table (GDT) and Relocations are fixed up so that RMM can run as position independent executable (PIE). The BSS is zero initialized which completes the C runtime initialization for cold boot.

2. **Platform initialization phase**

   The boot args are restored to their original registers and plat_setup() and plat_warmboot_setup() are invoked for cold and warm boot respectively. During cold boot, the platform is expected to consume the boot manifest which is part of the RMM-EL3 communications interface. The platform initializes any platform specific peripherals and also intializes and configures the translation table contexts for Stage 1.

3. **MMU enable phase**

   The EL2&0 translation regime is enabled after suitable TLB and cache invalidations.

4. **PAuth enable phase**

   Disable API, APK Trap, to allow PAuth instructions access from Realm without trapping. Initialize APIA Keys to random 128-bit value, Enable PAuth for R-EL2.

5. **RMM Main phase**

   Any cold boot or warm initialization of RMM components is done in this phase. This phase also involves invoking suitable EL3 services, like acquiring platform attestation token for Realm attestation.

After all the phases have completed successfully, RMM issues `RMM_BOOT_COMPLETE` SMC. The next entry into RMM from EL3 would be for handling RMI calls and hence the next intruction following the SMC call branches to the main SMC handler routine.

4.6 RMM-EL3 communication specification

The communication interface between RMM and EL3 is specified in RMM-EL3 communications interface specification in the TF-A repository.

CHAPTER

FIVE

SECURITY

5.1 Threat Model

5.1.1 Introduction

Threat modeling is an important part of Secure Development Lifecycle (SDL) that helps us identify potential threats and mitigations affecting a system.

In the next sections, we present the Threat Model for RMM, giving first a description of the target of evaluation using a data flow diagram. Then we provide a list of threats that we have identified based on the data flow diagram and potential threat mitigations.

5.1.2 Data Flow Diagram

This section describes the Data Flow Diagram for RMM.

Target of Evaluation

In this threat model, the target of evaluation is the Realm Management Monitor (RMM) in a system context for an Arm A-Class CPU with Realm Management Extension, as shown on Figure 1. Everything else on Figure 1 is outside of the scope of the evaluation.

RMM can be configured in various ways. In this threat model we consider only the most basic configuration. To that end we make the following assumptions:

- RMM image is run from either ROM, on-chip trusted SRAM or off-chip DRAM. Any memory shared with EL3 Firmware is located inside on-chip trusted SRAM. If RMM runs from off-chip DRAM, then RMM is vulnerable to DRAM attacks (such as rowhammer) and attacks which can probe and tamper off-chip memory.
- No experimental features are enabled. We do not consider threats that may come from them.
- RME hardware threats and threats covered by the RMM ABI will be covered in a dedicated Security Risk Analysis document (to be published in the future). Although there is some overlap with threats mitigated by RME hardware and RMM ABI, this threat model focuses on covering threats specific to the RMM implementation and associated data flows.

Data Flow Diagram

Figure 1 shows a high-level data flow diagram for RMM. The diagram shows a model of the different components of a RMM system and their interactions with other FW/SW components. A description of each diagram element is given in Table 1. In the diagram, the red broken lines indicate trust boundaries. Components outside of the broken lines are considered untrusted by RMM. Components inside the broken lines must be trusted by RMM, as they provide security foundations for its functionality.

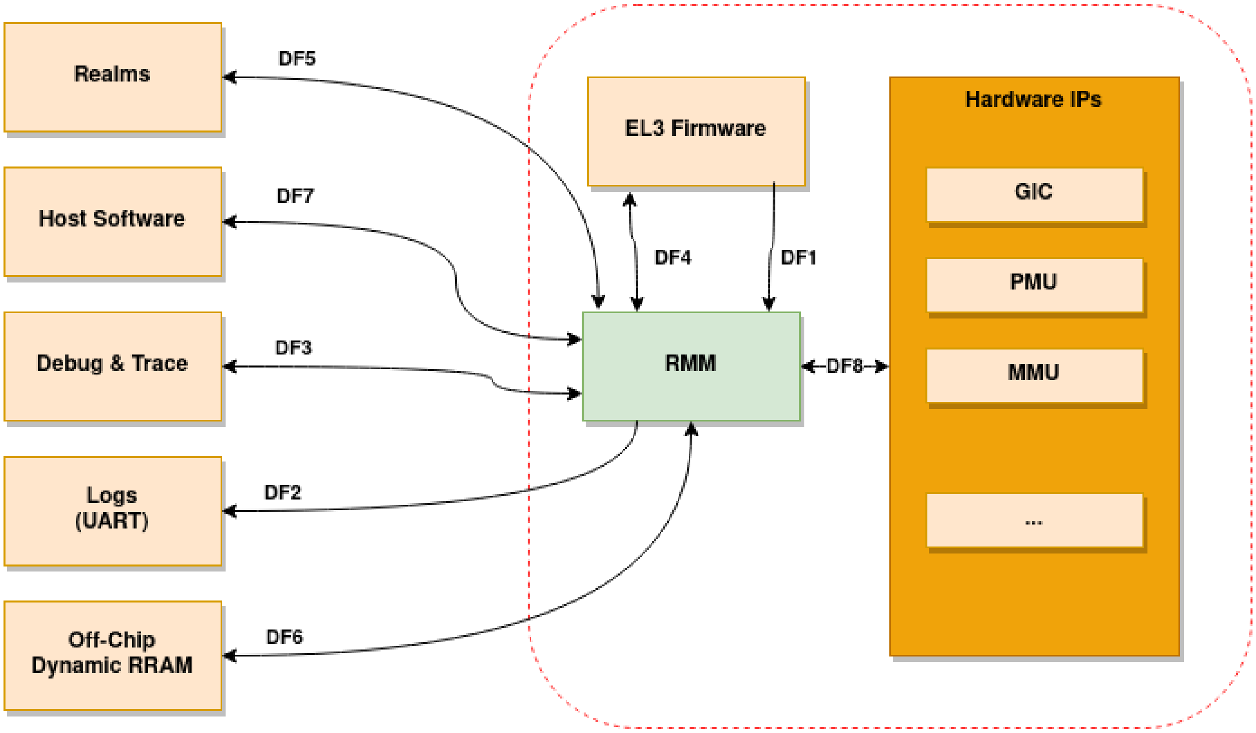


Table 1: Table 1: RMM Data Flow Diagram Description

Diagram Element	Description
DF1	At boot time, EL3 Firmware configures RMM through parameters stored in registers x0 to x3. It also passes a Boot Manifest using secure shared memory.
DF2	RMM log system framework outputs debug messages over a UART interface.
DF3	Debug and trace IP on a platform can allow access to registers and memory of RMM.
DF4	Interface for RMM-EL3 communication as per documented in RMM-EL3 Runtime Interface. RMM trusts EL3, which is part of a trusted subsystem.
DF5	Realm software can interact with RMM to request services through the RSI (Realm Service Interface). This also includes the PSCI interface.
DF6	Regardless of the type of memory from where RMM is executed, off-chip dynamic RAM (considered Non-Secure) may be used to store large data structures. This memory might be subject to different attacks.
DF7	NS Host interacts with RMM by issuing SMCs that are then forwarded from EL3 Firmware to RMM.
DF8	This path represents the interaction between RMM and various hardware IPs such as MMU controller and GIC. At boot time, RMM configures/initializes the IPs and interacts with them at runtime through interrupts and registers.

5.1.3 Threat Analysis components

In this section we identify all the possible *actors* involved in the Threat Model.

For each threat, we will identify the *asset* that is under threat, the *threat agent* and the *threat type*. Each threat will be given a *risk rating* that represents the impact and likelihood of that threat.

Any threat which needs to be mitigated by the RME hardware as well as by EL3 Root code is out of the scope of this Threat Model.

Assets

We have identified the following assets for RMM:

Table 2: Table 2: RMM Assets

Asset	Description
Sensitive Data	These include sensitive RMM and Realm data that an attacker must not be able to tamper with. Also, RMM should protect the confidentiality of of such data.
Code Execution	This represents the requirement that Realms should only run code in R-EL1/EL0 that is allowed by the RMM ABI. The Realm code execution cannot be hijacked by an attacker. This also represents the requirement that RMM should protect itself from privilege escalation and code injection by an attacker into R-EL2. The RMM execution cannot be hijacked by an attacker through the use of the RMM ABI.
Availability	Availability of Realm world is out of scope for this Threat Model. However, RMM should be designed in such a way that neither Realm nor RMM should significantly affect the availability of NS Host and Secure World.

Threat Agents

To understand the attack surface, it is important to identify potential attackers, i.e. attack entry points. The following threat agents are in scope of this threat model.

Table 3: Table 3: Threat Agents

Threat Agent	Description
RealmCode	Malicious or faulty code running in the Realm world, including R-EL0 and R-EL1 levels.
HostSoftware	Malicious or faulty code running in the Secure or Non-Secure world, EL0 and EL1 levels.
AppDebug	Physical adversary using debug build of RMM or access to debug sources of the system.

There is also a number of Threat Agents which are not covered by this Threat Model. These are listed in the table below.

Table 4: Table 4: Threat Agents not covered by this Threat Model

Threat Agent	Description
RootCode	Malicious or faulty code running at Root Level (e.g. EL3 Firmware). Since RootCode is part of TCB of the system, any fault in Root code is usually a critical vulnerability and not easily mitigated by RMM. This Threat is considered out-of-scope of analysis.
PhysicalAccess	Physical adversary having access to external device communication bus and to external flash communication bus using common hardware. An advanced physical atacker that has the capability to tamper with hardware (e.g. “rewiring” a chip using a focused ion beam -FIB- workstation or decapsulate the chip using chemicals).

Threat Types

In this threat model we categorize threats using the STRIDE threat analysis technique. In this technique a threat is categorized as one or more of these types: `Spoofing`, `Tampering`, `Repudiation`, `Information disclosure`, `Denial of service` or `Elevation of privilege`.

Threat Risk Ratings

For each threat identified, a risk rating that ranges from *informational* to *critical* is given based on the likelihood of the threat occurring if a mitigation is not in place, and the impact of the threat (i.e. how severe the consequences could be). Table 4 explains each rating in terms of score, impact and likelihood.

Table 5: Table 4: Rating and score as applied to impact and likelihood

Rating (Score)	Impact	Likelihood
Critical (5)	Extreme impact to entire organization if exploited.	Threat is almost certain to be exploited. Knowledge of the threat and how to exploit it are in the public domain.
High (4)	Major impact to entire organization or single line of business if exploited	Threat is relatively easy to detect and exploit by an attacker with little skill.
Medium (3)	Noticeable impact to line of business if exploited.	A knowledgeable insider or expert attacker could exploit the threat without much difficulty.
Low (2)	Minor damage if exploited or could be used in conjunction with other vulnerabilities to perform a more serious attack	Exploiting the threat would require considerable expertise and resources
Informational (1)	Poor programming practice or poor design decision that may not represent an immediate risk on its own, but may have security implications if multiplied and/or combined with other threats.	Threat is not likely to be exploited on its own, but may be used to gain information for launching another attack

Aggregate risk scores are assigned to identified threats. Specifically, the impact score multiplied by the likelihood score. For example, a threat with high likelihood and low impact would have an aggregate risk score of eight (8); that is, four (4) for high likelihood multiplied by two (2) for low impact. The aggregate risk score determines the finding's overall risk level, as shown in the following table.

Table 6: Table 5: Overall risk levels and corresponding aggregate scores

Overall Risk Level	Aggregate Risk Score (Impact multiplied by Likelihood)
Critical	20–25
High	12–19
Medium	6–11
Low	2–5
Informational	1

The likelihood and impact of a threat depends on the target environment in which RMM is running. For example, attacks that require physical access are unlikely in server environments while they are more common in Internet of Things (IoT) environments. In this threat model we only consider `Server` target environments.

5.1.4 Threat Assessment

The following threats were identified by applying STRIDE analysis on each diagram element of the data flow diagram. The threats are related to software and implementation specific to TF-RMM.

For each threat, we strive to indicate whether the mitigations are currently implemented or not. However, the answer to this question is not always straightforward. Some mitigations are partially implemented in the generic code but also rely on the platform code to implement some bits of it. This threat model aims to be platform-independent and it is important to keep in mind that such threats only get mitigated if the platform properly fulfills its responsibilities.

Also, some mitigations might require enabling specific features, which must be explicitly turned on via a build flag.

The `Pending actions?` box highlights any action that needs to be done in order to implement the mitigations.

ID	01
Threat	Information leak via UART logs. During the development stages of software it is common to print all sorts of information on the console, including sensitive or confidential information such as crash reports with detailed information of the CPU state, current registers values, privilege level or stack dumps. This information is useful when debugging problems before releasing the production version but it could be used by an adversary to develop a working exploit if left enabled in the production version. This happens when directly logging sensitive information and more subtly when logging based on sensitive data that can be used as a side-channel information by an adversary.
Diagram Elements	DF2
Assets	Sensitive Data
Threat Agent	AppDebug
Threat Type	Information Disclosure
Impact	Informational (1)
Likelihood	Informational (1)
Total Risk Rating	Informational (1)
Mitigations	1) For production releases: i) Remove sensitive information logging. ii) Do not conditionally log based on sensitive data. iii) Do not log high precision timing information. iv) Do not log register contents which may reveal secrets during crashes (Error Syndrome registers are allowed to be logged). 2) Provide option to fully disable RMM logging for production releases.
Mitigations implemented?	1) Yes/Platform-specific. The default log level does not output verbose log. RMM does not implement crash reporting. Messages produced by the platform code should use the appropriate level of verbosity so as not to leak sensitive information in production builds. 2) Yes. RMM provides the `LOG_LEVEL` build option which can be used to disable all logging.

ID	02
Threat	An adversary can try stealing information by using RMM ABI. NS Host accesses RMM through RMM ABI. Malicious code can attempt to invoke services that would result in disclosing private information or secrets.
Diagram Elements	DF7
Assets	Sensitive Data
Threat Agent	HostSoftware
Threat Type	Information Disclosure
Impact	High (4)
Likelihood	High (4)
Total Risk Rating	High (16)
Mitigations	1) Ensure that RMM protects the Realm memory by using GPT service provided by EL3 Firmware and appropriate Stage 2 protections. NS Host must not be able to change or access Realm memory. 2) NS host must not be able to change or access Realm CPU register contents other than what is allowed by RMM ABI. Root code should perform proper context switching of certain subset of CPU registers as mandated in RMM-EL3 Communication Interface when entering and exiting the Realm world. Similarly, RMM should context switch any registers not managed by EL3 when entering/exiting Realms. 3) The RME Architecture provides means to configure memory isolation to the Realm world. RMM relies on Root code for correct RME setup. But when undelegating memory to the Normal world, RMM needs to ensure that suitable memory scrubbing is implemented. Also, RMM should ensure any architectural caches are invalidated before returning back to NS Host.
Mitigations implemented?	1) Yes. 2) Yes. 3) Yes.
Pending actions?	None.

ID	03
Threat	An adversary can perform a denial-of-service attack on the system by causing the Realm world/RMM to deadlock, crash or enter into an unknown state.
Diagram Elements	DF5, DF7
Assets	Availability
Threat Agent	RealmCode, HostSoftware
Threat Type	Denial of Service
Impact	Medium (3)
Likelihood	Low (2)
Total Risk Rating	Medium (6)
Mitigations	1) Upon an unrecoverable/catastrophic condition, RMM should issue a `panic()`. This would return to EL3 Firmware, keeping the availability of the overall system. It would be EL3 responsibility to decide how to proceed (e.g. by disabling the whole Realm world). 2) EL3 Firmware needs to implement a watchdog mechanism to recover CPUs from Realm World.
Mitigations implemented?	1) No. 2) Mitigation would need support from EL3 Firmware.
Pending actions?	`panic()` needs appropriate implementation to return to EL3 Firmware.

ID	04
Threat	Malicious Host or Realm code can attempt to place the RMM into an inconsistent state due to incorrect implementation of RMM state machines. This inconsistency can be exploited to lead incorrect operation of RMM.
Diagram Elements	DF5, DF7
Assets	Availability, Sensitive Data, Code Execution
Threat Agent	RealmCode, HostSoftware
Threat Type	Denial of Service, Tampering, Elevation of privilege, Information Disclosure
Impact	Medium (3)
Likelihood	Low (2)
Total Risk Rating	Medium (6)
Mitigations	1) State machines should be tested for all the transitions and validated that all invalid transitions and inputs are rejected. 2) The RMM ABI mandates pre and post conditions for each ABI. The tests should verify that these conditions are adhered to and implemented. 3) Static analyzers and model checkers can be used to uncover bugs in implementation. 4) Fuzz testing can be employed to uncover further issues in implementation. 5) Upon an unrecoverable/catastrophic condition occurs, RMM should issue a `panic()` to prevent further corruption of data or propagation of errors.
Mitigations implemented?	1) Partial. There are various tests in TFTF, ACS and kvm-unit-tests for exercising the ABI which triggers the state machines. Unit tests are also present for some components to exercise internal APIs which can further test conditions and invalid cases which cannot be triggered via RMM ABI. 2) Partial. Code reviews to ensure the implementation complies the required conditions. Automated checking via CBMC to validate the same is also being implemented. 3) Yes. CPPCheck and Coverity scan are used to detect issues. CBMC is also utilized as a model checker. 4) No. 5) Yes.
Pending actions?	

ID	05
Threat	Malicious Host or Realm code can attack RMM by calling unimplemented SMC calls or by passing invalid arguments to the ABI.
Diagram Elements	DF5, DF7
Assets	Sensitive Data, Code Execution
Threat Agent	RealmCode, HostSoftware
Threat Type	Denial of Service, Tampering, Elevation of privilege, Information Disclosure
Impact	High (4)
Likelihood	High (4)
Total Risk Rating	High (16)
Mitigations	1) Validate SMC function IDs and arguments before using them. 2) Invalid/Unimplemented SMCs should return back to caller with error code. 3) Tests to exercise invalid arguments and unimplemented SMCs.
Mitigations implemented?	1) Yes. 2) Yes. 3) Partial. The ACS test utility exercises many invalid inputs. Unit tests also test various invalid cases.
Pending actions?	Expand unit tests to cover the RMM ABI interface and test for invalid inputs.

ID	06
Threat	An adversary can make use of incorrect implementation of concurrent sections in RMM to cause data corruption or dead/live locks.
Diagram Elements	DF5, DF7
Assets	Availability, Sensitive Data, Code Execution
Threat Agent	RealmCode, HostSoftware
Threat Type	Denial of Service, Tampering, Elevation of privilege, Information Disclosure
Impact	Medium (3)
Likelihood	Low (2)
Total Risk Rating	Medium (6)
Mitigations	1) Follow locking discipline described in RMM Locking Guidelines when implementing concurrent sections in RMM. 2) Validate locking discipline using tests which can run multiple threads in RMM. 3) Fuzz tests on RMM with multiple threads.
Mitigations implemented?	1) Yes. 2) Yes. The TFX test has tests which can test concurrent sections in RMM. Also, stress tests in CI will also test this scenario. 3) No. Need further investigation.
Pending actions?	Enhance TFX tests to test more concurrent sections in RMM. Investigate the possibility of multithread Fuzz Testing.

ID	07
Threat	A Realm can claim to be another Realm. NS Host can associate the PA of one Realm to another Realm.
Diagram Elements	DF5, DF7
Assets	Sensitive Data
Threat Agent	RealmCode, HostSoftware
Threat Type	Spoofing
Impact	High (4)
Likelihood	Low (2)
Total Risk Rating	Medium (8)
Mitigations	1) A Realm should not be able to spoof another realm. The NSHost must not be able to assign a granule/metadata to a Realm which is already assigned to another Realm.
Mitigations Implemented?	1) Yes. This mitigation is inherently supported by the RMM ABI. SMC call from realm is always associated to the Realm Descriptor (RD) and the RMM ABI does not allow spoofing of RD. NS Host always has to pass the address of a valid RD to make requests to the corresponding Realm. RMM maintains a global granule array and every granule linked to a Realm has a specific State and reference count associated with it. Hence, the NS Host cannot associate an already Realm associated granule to another Realm.
Pending actions?	None.

ID	08
Threat	An adversary could execute arbitrary code, modify some state variables, change the normal flow of the program or leak sensitive information if memory overflows and boundary checks when accessing resources are not properly handled. In the particular case in which the control flow can be changed by a stack overflow, code execution can also be subverted by an adversary. Like in other software, RMM has multiple points where memory corruption and security errors can arise. Some of the errors include integer overflow, buffer overflow, incorrect array boundary checks and incorrect error management. Improper use of asserts instead of proper input validations might also result in these kinds of errors in release builds.
Diagram Elements	DF5, DF7
Assets	Code Execution, Sensitive Data, Availability
Threat Agent	RealmCode, HostSoftware
Threat Type	Tampering, Information Disclosure, Elevation of Privilege
Impact	Medium (3)
Likelihood	Low (2)
Total Risk Rating	Medium (6)
Mitigations	1) Use proper input validation. 2) Enable Architecture security features to mitigate buffer overflow and ROP/JOP issues. 3) Utilize stack protection mechanism provided by the compiler. 4) Design suitable per CPU stack protection, so another CPU cannot corrupt stack which does not belong to it. 5) Suitable testing to test bounds of inputs. 6) Employ secure coding guidelines like MISRA to remove many of the type safety issues associated with the C language. 7) Use static analyzers to check for common issues. Also, make use of model checkers to validate loop bounds and other bounds in the source code.
Mitigations implemented?	1) Yes. 2) Yes. RMM Enables many Architecture security features like PAuth and BTI but there is ongoing action to enable more architectural

ID	09
Threat	SMC calls can leak sensitive information from RMM memory via microarchitectural side channels. Microarchitectural side-channel attacks such as Spectre can be used to leak data across security boundaries. An adversary might attempt to use this kind of attack to leak sensitive data from RMM memory. Also, some SMC calls, such as the ones involving encryption when applicable, might take different amount of time to complete depending upon the parameters. An adversary might attempt to use that information in order to infer secrets or to leak sensitive information.
Diagram Elements	DF5, DF7
Assets	Sensitive Data
Threat Agent	RealmCode, HostSoftware
Threat Type	Information Disclosure
Impact	Medium (3)
Likelihood	Informational (1)
Total Risk Rating	Low (3)
Mitigations	1) Enable appropriate speculation side-channel mitigations as recommended by the Architecture. 2) Enable appropriate timing side-channel protections available in the Architecture. 3) Ensure the software components dealing with sensitive data use Data Independent algorithms. 4) Ensure that only required memory is mapped when executing a Realm or doing operations in RMM so as to minimize effects of CPU speculation.
Mitigations implemented?	1) Yes. 2) Yes. `FEAT_DIT` is enabled for RMM. 3) Yes. RMM relies on MbedTLS library to use algorithms which are data independent when handling sensitive data. 4) Yes. The slot buffer design for dynamically mapping memory ensures that only required memory is mapped into RMM.
Pending actions?	Review speculation vulnerabilities and ensure RMM

ID	10
Threat	Misconfiguration of the S2 MMU tables may allow Realms to access sensitive data belonging to other Realms or incorrect mapping of NS memory may allow Realms to corrupt NSHost memory.
Diagram Elements	DF5, DF7
Assets	Sensitive Data, Code execution
Threat Agent	RealmCode, HostSoftware
Threat Type	Information Disclosure
Impact	High (4)
Likelihood	Low (2)
Total Risk Rating	Medium (8)
Mitigations	1) Ensure proper implementation of S2 table management code in RMM. It should not be possible to trigger misconfiguration of S2 tables using RMM ABI. Appropriate tests to ensure that the implementation is robust. 2) The RMM specification mandates the rules for assigning memory to a Realm and IPA management. Ensure the rules mandated by the RMM specification are validated by suitable tooling.
Mitigations implemented?	1) Partially. There are various tests like kvm-unit-tests, TFTF, TFX and ACS to test the implementation. Unit tests of S2 tables need to be implemented. Static analysis is in place to detect issues. 2) Partially. Code reviews to ensure the implementation complies with the required conditions. Automated checking via CBMC to validate the same is also being implemented.
Pending actions?	Increase testing coverage of S2 table management code in RMM. Integrate CMBC into RMM testing.

ID	11
Threat	Realm code flow diversion through REC metadata manipulation from Host Software. An adversary with access to a Realm's REC could tamper with the structure content and affect the Realm's execution flow.
Diagram Elements	DF7
Assets	Code Execution
Threat Agent	HostSoftware
Threat Type	Tampering
Impact	High (4)
Likelihood	Low (2)
Total Risk Rating	Medium (8)
Mitigations	1) The RMM specification mandates that sensitive metadata like REC should be stored in Realm PAS. Also, the specification does not allow NSHost to manipulate REC contents via RMI once the associated Realm is Activated. Ensure that the RMM specification guidelines are enforced. 2) Map sensitive metadata into RMM S1 tables only when manipulating the Realm/REC. Once RMM is finished manipulating the metadata, unmap it from S1 tables. Thus the time window when RMM can access the metadata is kept to a minimum thus reducing the opportunity to corrupt the metadata.
Mitigations implemented?	1) Yes. 2) Yes. The slot-buffer mechanism in RMM is used to map metadata only when needed and it is unmmaped when the access is not required.
Pending actions?	None

ID	12
Threat	Use of PMU and MPAM statistics to increase the the accuracy of side channel attacks.
Diagram Elements	DF5, DF7
Assets	Sensitive Data
Threat Agent	RealmCode, HostSoftware
Threat Type	Information Disclosure
Impact	Medium (3)
Likelihood	Informational (1)
Total Risk Rating	Low (3)
Mitigations	1) Save/Restore performance counters on on transitions between security domains or between Realms. 2) Configure MPAM so that is either disabled or set to default values for Realm world.
Mitigations implemented?	1) PMU is saved and restored. 2) MPAM is not enabled for Realm world.
Pending actions?	None.

ID	13
Threat	Misconfiguration of the hardware IPs and registers may lead to data leaks or incorrect behaviour that could be damaging on its own as well as being exploited by a threatening agent. RMM needs to interact with several hardware IPs as well as system registers for which it uses its own libraries and/or drivers. Misconfiguration of such elements could cause data leaks and/or system malfunction.
Diagram Elements	DF5, DF7
Assets	Sensitive Data, Availability
Threat Agent	RealmCode, HostSoftware
Threat Type	Information Disclosure, Denial of Service
Impact	High (4)
Likelihood	Low (2)
Total Risk Rating	Medium (8)
Mitigations	1) Code reviews. 2) Testing on FVPs and other hardware and emulation platforms to check for correct behaviour.
Mitigations implemented?	1) Yes. 2) Yes. RMM is tested regularly on FVP and more platforms will be added in future as they become available.
Pending actions?	None

CHAPTER

SIX

RESOURCES

6.1 Application Notes

6.1.1 CBMC

CBMC is a Bounded Model Checker for C and C++ programs. For details see CBMC Home

CBMC in RMM

CBMC needs to be run on a C program that has a single entry point. To test all the RMM ABI commands, for each command a testbench file is created. These files are generated by a script offline from the RMM MRS (Machine Readable Specification), and committed to the RMM repository under the folder `tools/cbmc/testbenches`

Note: Currently only a subset of the ABI calls have a testbench implemented. Also there are errors reported by CBMC for some of the testbenches. Further testbenches and fixes are expected to be added later.

These files contain asserts that correspond to the Faliure and Success conditions defined in the RMM specification. To read on further how such a file should be defined please refer to Writing a good proof

The recommended way for installing CBMC is via the pre-built package found at the github release page. The same page contains the instructions for installing the different release packages.

An example install command for Ubuntu linux is

```
dpkg -i ubuntu-20.04-cbmc-5.95.1-Linux.deb
```

The invocation of CBMC tool is integrated in the RMM CMake system. CBMC analysis can be run by passing specific targets (detailed later) to the build command. to make the targets available the RMM build must be configured with `-DRMM_CONFIG=host_defcfg -DHOST_VARIANT=host_cbmc` options.

The results of a CBMC run are generated in the `${RMM_BUILD_DIR}/tools/cbmc/cbmc_${MODE}_results` directory. There are 3 files, `${TESTBENCH_FILE_NAME}.${MODE}.[cmd|error|output]` generated for each RMM ABI under test, each one containing the CBMC command line, the CBMC executable's output to the standard error, and the output to the standard out respectively. There is also a single `SUMMARY.${MODE}` file is generated for each build.

For an example build command please refer to *RMM Build Examples*

The CMake system by default runs CBMC on all the testbenches. This can take a long time, and it can be convenient to run a single testcase at once. This can be achieved using the option -DRMM_CBMC_SINGLE_TESTBENCH="testbench_name". The list of possible testbench_name``s can be listed by using the option ``-DRMM_CBMC_SINGLE_TESTBENCH="help".

The CMake system provides different modes in which CBMC can be called, along with their respective build targets.

CBMC Assert

In this mode CBMC is configured, so that it tries to find inputs that makes an assertion in the code to fail. If there is such an input, then CBMC provides a trace that leads to that assertion failure.

To use this mode the target `cbmc-assert` must be passed to the build command.

CBMC Analysis

In this mode CBMC is configured to generate assertions for certain properties in the code. The properties are selected so that for example no buffer overflows, or arithmetic overflow errors can happen in the code. For more details please refer to Automatically Generating Properties. Then CBMC is run in a configuration similar to the Assert mode, except that this time traces are not generated.

To use this mode the target `cbmc-analysis` must be passed to the build command.

CBMC Coverage

This mode checks whether all the conditions for an ABI function are covered. The pre and post conditions for the command are expressed as boolean values in the testbench, and a `__CPROVER_cover()` macro is added for each condition that is expressed with the pre and post conditions. CBMC is configured to try to generate an input for each `__CPROVER_cover()` call that makes the code reach that call.

To use this mode the target `cbmc-coverage` must be passed to the build command.

Note: For all the modes the summary files are committed in the source tree as baseline in `tools/cbmc/testbenches_results/BASELINE.${MODE}`.

Build The CBMC testbench with GCC

In the RMM CMake system there is an option to build the CBMC testbench with GCC compiler. The resulting binary doesn't have any particular value, however during the compilation GCC may flag errors that can cause CBMC work unexpectedly. For example functions that are defined in a file that is linked during the CBMC build, however not declared, due to a missing include. In this case CBMC seems to be silently ignoring the function body. This error is quite difficult to find using only CBMC output.

To use this mode the target `cbmc-gcc` must be passed to the build command.

cbmc-viewer

cbmc-viewer is a python package that can parse the XML output of CBMC. It generates a html report that can be opened in a browser. The report contains a collapsible representation of assert traces, and clickable links to the source code locations associated with a specific trace item.

The RMM cmake build system is capable of generating the cbmc-viewer report. If the option `-DRMM_CBMC_VIEWER_OUTPUT=ON` is passed to the RMM Cmake configuration command then the Cmake system calls cbmc-viewer and generates the report under `${RMM_BUILD_DIR}/tools/cbmc/cbmc_${MODE}_results/report`

Please note that the CMake build system currently only generates report for the cbmc-assert target. The cbmc-coverage and cbmc-analysis targets doesn't generate trace, so generating a report wouldn't be useful.

cbmc-viewer can be installed using the following command:

```
python3 -m pip install cbmc-viewer
```

For further details and installation guide on cbmc-viewer please see the cbmc-viewer github page.

CBMC proof debugger

CBMC proof debugger is an extension to a popular code editor that can be used to load the json summaries of a CBMC analysis that is generated by cbmc-viewer. The trace then can be explored in the built in debugger of the editor as if stepping through an actual code execution.

For further details on installing and using the extension please see CBMC proof debugger in the editor's extensions page.

CHAPTER

SEVEN

GLOSSARY

This glossary provides definitions for terms and abbreviations used in the RMM documentation.

You can find additional definitions in the Arm Glossary.

AArch64
64-bit execution state of the ARMv8 ISA

GPF
Granule Protection Fault

IPA
Intermediate Physical Address

PA
Physical Address

PAR
Protected Address Range

PAS
Physical Address Space

RD
Realm Descriptor

REC
Realm Execution Context

RMM
Realm Management Monitor

RTT
Realm Translation Table

TF-A
Trusted Firmware-A

TTE
Translation Table Entry

VHE
Virtualization Host Extensions

BIBLIOGRAPHY

[EWD310] Dijkstra, E.W. Hierarchical ordering of sequential processes. EWD 310.

[EWD625] Dijkstra, E.W. Two starvation free solutions to a general exclusion problem. EWD 625.

[MCS] Mellor-Crummey, John M. and Scott, Michael L. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM TOCS, Volume 9, Issue 1, Feb. 1991.

[WS2001] Stallings, W. (2001). Operating systems: Internals and design principles. Upper Saddle River, N.J: Prentice Hall.

INDEX