Realm Management Monitor

TF-RMM Contributors

Nov 22, 2022

CONTENTS

CHAPTER

ONE

ABOUT

1.1 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 :

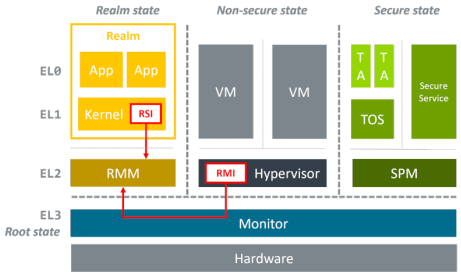


Figure 1. Realm VM execution

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, and QCBOR.

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 Dan Handley <dan.handley@arm.com>

GitHub ID danh-arm

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

GitHub ID soby-mathew

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

GitHub ID javier-almansasobrino

1.3 Change-log and Release notes

1.3.1 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*.

1.3.2 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

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

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.

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

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 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
```

2.7.1 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.

2.8 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/#download or from the official github https://github.com/danmar/cppcheck/

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

```
sudo mkdir /usr/local/share/Cppcheck/misra
sudo cp -a <path to the misra rules file>/<file name> /usr/local/share/Cppcheck/misra/
↪misra.rules
```

2.9 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}
```

2.10 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.

2.11 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 -DRMM_STATIC_ANALYSIS_CPPCHECK=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 CERT_C/MISRA/THREAD SAFETY (example with MISRA):

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

2.12 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_SIZE		4096	Granule Size used by RMM
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		40	Log level to apply for RMM (0 - 50)
RMM_STATIC_ANALYSIS			Enable static analysis checkers
RMM_STATIC_ANALYSIS_CPPCHECK	ON | OFF	ON	Enable Cppcheck static analysis
RMM_STATIC_ANALYSIS_CPPCHECK_SEI_CERT	ON | OFF	ON	Enable Cppcheck's SEI CERT C checker
RMM_STATIC_ANALYSIS_CPPCHECK_MISRA	ON | OFF	OFF	Enable Cppcheck's MISRA C:2012 checker
RMM_STATIC_ANALYSIS_CPPCHECK_THREAD_SAFETY	ON | OFF	ON	Enable Cppcheck's thread safety checker
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_MAX_MMAP_REGIONS		5	Maximum number of mmap regions to be allocated for the platform
RMM_NUM_PAGES_PER_STACK		3	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

2.13 RMM LLVM Build

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

2.14 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.

- A line must be <= 80 characters, except for string literals as it would make any search for it more difficult.

- A variable should not be longer than 31 characters.

  Although the C11 specification specifies that the number of significant 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	80 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

```
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.

```
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);
        } 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.
  - For other pointer arguments in a function declaration, use `void *`. This includes pointers to types that are abstracted away from the known API and pointers to arbitrary data. This allows the calling function to pass a pointer argument to the function without any explicit casting (the cast to `void *` is implicit). The function implementation can then do the appropriate casting to a specific type.
  - 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 `u_register_t` when it's important to store the contents of a register in its native size (64-bit in *AArch64*). This is not a standard *C11* type but is widely available in libc implementations. Where possible, cast the variable to a more appropriate type before interpreting the data. For example, the following structure uses this type to minimize the storage required for the set of registers:

```
typedef struct aapcs64_params {
        u_register_t arg0;
        u_register_t arg1;
        u_register_t arg2;
        u_register_t arg3;
        u_register_t arg4;
        u_register_t arg5;
        u_register_t arg6;
```

(continues on next page)

(continued from previous page)

```
        u_register_t arg7;
} aapcs64_params_t;
```

If some code wants to operate on `arg0` and knows that it represents a 32-bit unsigned integer on all systems, cast it to `unsigned int`.

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

Currently the RMM implementation conforms to the RMM Beta0 Specification which means it is not yet ready to be productised.

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: ...
```

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:

```
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`
 * 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);

/*
```

(continues on next page)

(continued from previous page)

```
 * 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,
                               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
```

(continues on next page)

(continued from previous page)

```
 * 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 RMM Folder and Component organization

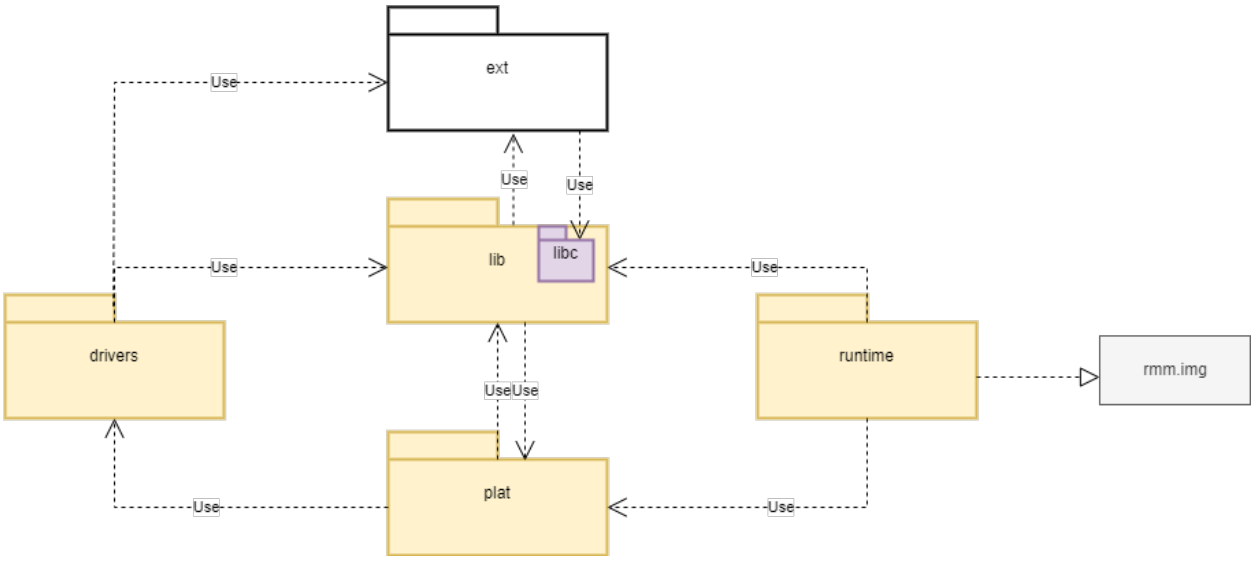
4.2.1 Root Level Folders and Components

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

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

4.2.2 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")

# 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)
```

4.3 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.

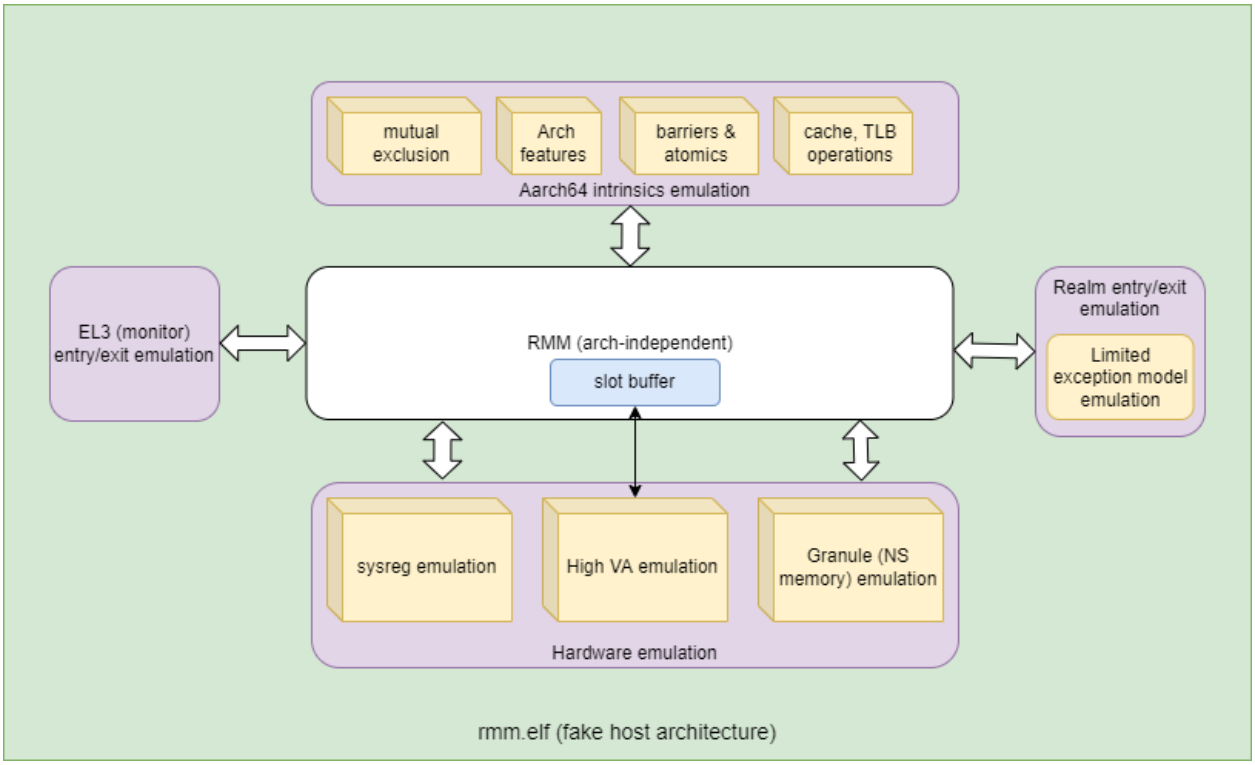
We expect the fake host architecture to be developed over time in future to cover some of the employment scenarios described above. The current code may not reflect the full scope of this architecture as discussed in this document.

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.

4.3.1 Fake host architecture design



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.

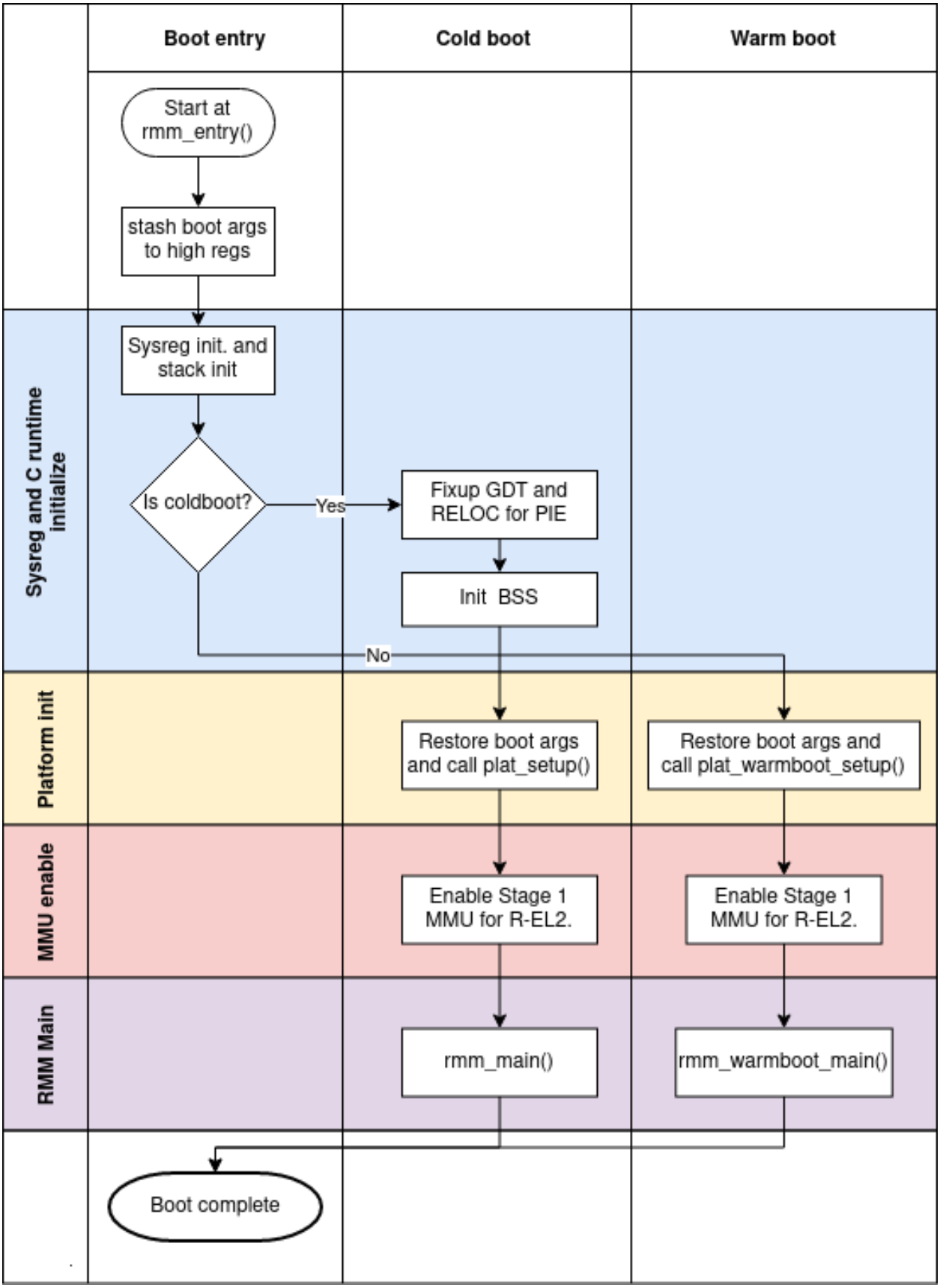
4.3.2 Fake host architecture employment scenarios implemented or ongoing

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.

4.4 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
No
Fixup GDT and RELOC for PIE
Init BSS
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.
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. **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.

4.5 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

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

PAR Protected Address Range

PAS Physical Address Space

RD Realm Descriptor

REC Realm Execution Context

RMM Realm Management Monitor

TF-A Trusted Firmware-A

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