Neoverse Reference Design Platform Software

unknown

Mar 19, 2025

ABOUT

CHAPTER

ONE

REFERENCE DESIGN

A Reference Design (RD) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP.

Specifically, Neoverse reference designs provide resources with best practices on how to integrate a Neoverse compute subsystem within a larger SoC. These compute subsystems are targeted at addressing requirements for applications in the cloud-to-edge infrastructure markets.

Neoverse products are categorized as follows:

- Neoverse V-Series: Maximum Performance
- Neoverse N-Series: Scale Out Performance
- Neoverse E-Series: Efficient Throughput

Refer to Arm Neoverse to learn more about the intended use cases for this products.

The Neoverse reference designs are also available as fixed virtual platform models and when used with a software stack, they provide a way to explore the features available in the reference design compute subsystem through software.

We provide a companion software stack for each of the Neoverse reference designs and the source code can be found in Arm's Gitlab repository, located at https://gitlab.arm.com/infra-solutions/reference-design

This documentation lists the available reference designs, the features each support and how to interact with them.

Note: A reference design is also referred to as platform.

Table below provides the user an overview of Neoverse reference designs, and their current status.

Platform	Status
RD-V3-R1 Cfg1	Active
RD-V3-R1	Active
RD-V3 Cfg2	Active
RD-V3 Cfg1	Active
RD-V3	Active
RD-V2	Maintenance
RD-N2 Cfg3	Maintenance
RD-N2 Cfg2	Maintenance
RD-N2 Cfg1	Maintenance
RD-N2	Maintenance
RD-V1	Legacy
RD-V1 MC	Legacy
RD-N1 Edge	Legacy
RD-N1 Edge X2	Legacy
SGI-575	Legacy

Visit the *Getting Started* chapter to learn how to setup an host machine to download, compile and execute a platform feature (i.e.: boot an OS), on a fixed virtual platform.

For questions about the Neoverse Reference Design platform software stack, write to support@arm.com.

CHAPTER

TWO

SOFTWARE STACK

The Neoverse software stack integrates multiple software components to provide a reference implementation of a software solution that can be used demonstrate various capabilities of the respective platform.

A typical software stack is illustrated in Fig. 2.1.

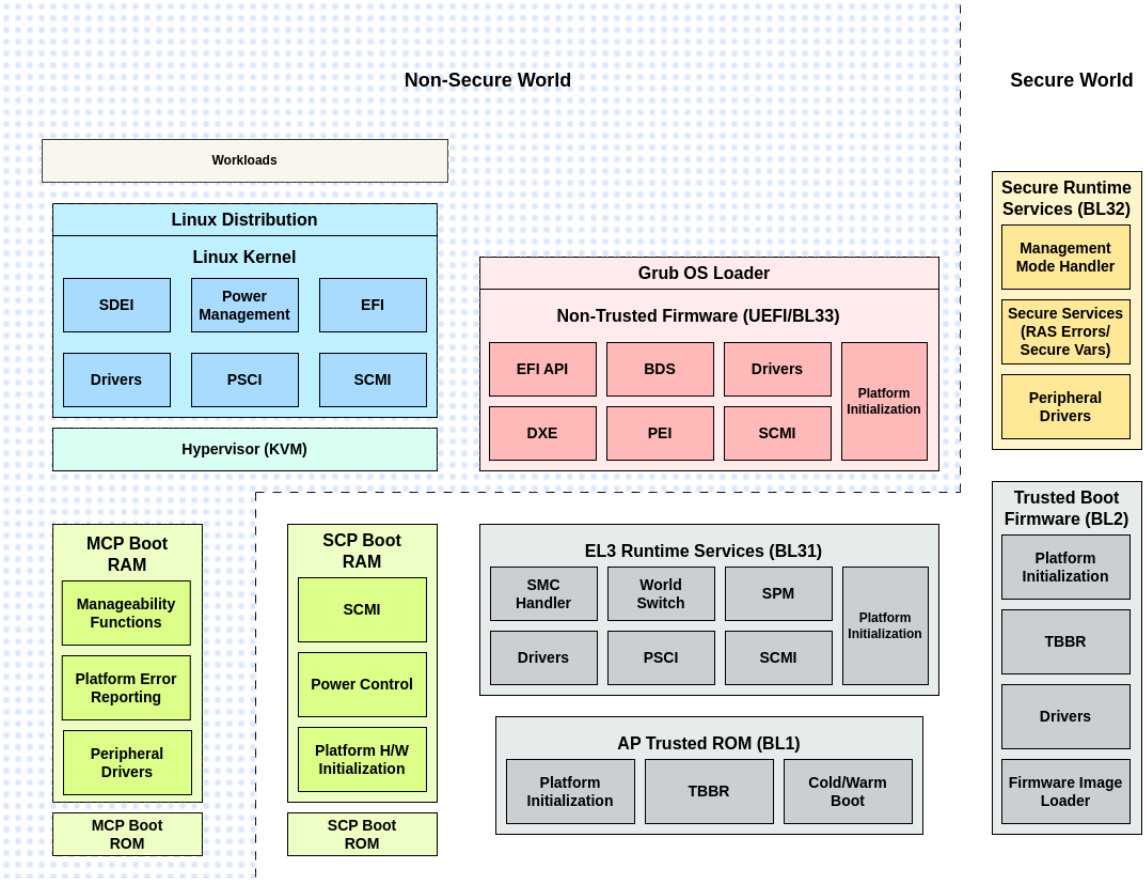


Fig. 2.1: High-level software illustration of a Neoverse Reference Design.

Some reference designs support the Realm Management Extension (RME), and their software stack is ilustrated in Fig. 2.2.

The following sections list the various software components that are included the Neoverse software stack.

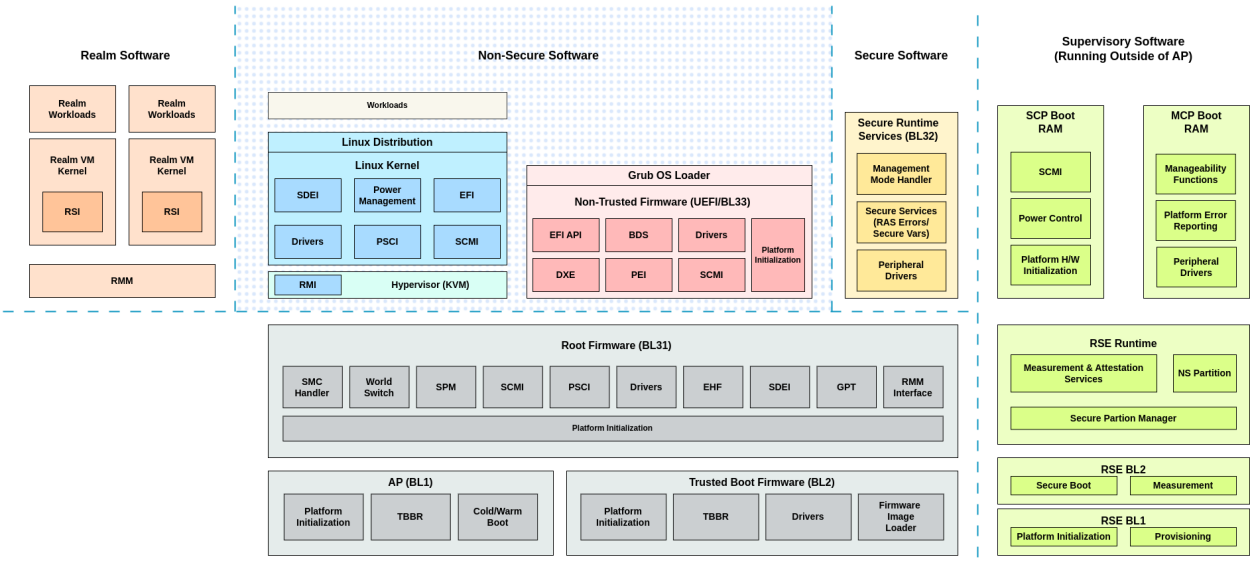


Fig. 2.2: High-level software illustration of a Neoverse Reference Design with RME.

2.1 MSCP Firmware

Neoverse reference design platforms include a System Control Processor (SCP) sub-system and a Manageability Control Processor (MCP) sub-system. The SCP sub-system is tasked with the management of system clocks, power control, configuring the system interconnect, memory controllers, PCIe controllers and many other functionalities. The MCP sub-system is tasked with the management of communications with an external Baseboard Management Controller (BMC). The firmware executed by the SCP and MCP processors is sourced from the SCP-firmware open-source project.

2.2 Trusted Firmware

Trusted Firmware-A (TF-A) software component provides a reference open-source implementation of a secure monitor executing at EL3 exception level. It implements various Arm interface standards including the Power State Coordination Interface (PSCI), Trusted Board Boot Requirements (TBBR), SMC Calling Convention, System Control and Management Interface and others. The trusted firmware executes in various stages - Boot Loader stage 1 (BL1) AP Trusted ROM, Boot Loader stage 2 (BL2) Trusted Boot Firmware, Boot Loader stage 3-1 (BL3-1) EL3 Runtime Firmware, Boot Loader stage 3-2 (BL3-2) Secure-EL1 Payload (optional) and Boot Loader stage 3-3 (BL3-3) Non-trusted Firmware.

2.3 EDK2

EFI Development Kit 2 (edk2) is a firmware development environment for the UEFI and PI specifications. UEFI is a specification that defines an interface between the firmware and an Operating System (OS). UEFI defines the firmware interfaces and boot services that are required for booting a standards-based OS. UEFI also defines run-time services, for example, time, variable that an OS can invoke at runtime. The reference design platform stack integrates both the edk2 and edk2-platforms open-source projects to support an implementation of EFI API for the platform.

2.4 Linux Kernel

Linux kernel is used as the host operating system kernel for the reference design platforms. ACPI tables are used to describe the platform to the linux kernel. All the capabilities of the linux kernel are used to demonstrate the various functionalities of the platform software including power management, device assignment, RAS and many others.

2.5 Other software components

The platform software stack uses the following additional software components to provide an integrated software solution for the Neoverse reference design platforms.

- Trusted Firmware Test Framework
- Grub
- Busybox
- Buildroot
- ACPICA
- Mbed TLS
- ACS
- EFI Tools
- UEFI SCT

CHAPTER

THREE

REPO TOOL & MANIFESTS

repo tool is a wrapper around the git version control system to manage multiple git repositories, it simplifies several git operations, in particular the *sync* (download) of software sources to a local environment.

To achieve this, repo uses a manifest file, which is a collection of repositories with some attributes to specify for example the revision of a particular repository or the local path that repository will have once downloaded to our local environment.

3.1 Manifest File in Detail

A manifest file is written in Extensible Markup Language (XML), for a full list of supported elements and attributes please see repo manifest format.

Take the excerpt of an existing manifest file in Neoverse reference design gitlab repository to understand how manifests are defined.

```
1 <manifest>
2   <remote fetch="https://git.gitlab.arm.com/infra-solutions/reference-design/" name="arm
  ↪"/>
3   <remote fetch="https://github.com/" name="github"/>
4
5   <project remote="arm" name="platsw/scp-firmware" path="scp" revision="refs/tags/RD-
  ↪INFRA-2024.12.20"/>
6 </manifest>
```

Line #2, the element `remote` defines two attributes, `fetch` and `name`.

- **fetch** is the base url of a git repository.
- **name** is an alias for that repository.

Line #5, the element `project` also defines the attributes `name`, `path` and `revision`.

- **name** is the name of the project in the git repository.
- **path** is the local path where the project will be cloned to, relative to where the manifest is initialised.
- **revision** is the git revision to clone, this can be a *branch*, a *tag* or a *commit sha id*.

A remote can have multiple repositories, the link from a remote to a project is given by the alias set in line #2 and used in line #5 as `remote="arm"`.

The `revision` attribute when set to *tag* or a *commit sha id* is what makes the software stack being reproducible because it will fetch the same software sources consistently. More details provided in the section *Pinned vs Non-Pinned*.

Translating the syntax of a manifest file to git, the underlying command of line #5 to run would be:

```
git clone --branch refs/tags/RD-INFRA-2024.12.20 https://git.gitlab.arm.com/infra-
↪solutions/reference-design/platsw/scp-firmware ./scp
```

3.2 Manifest (Pinned vs Non-Pinned)

Now that the user knows how a manifest file is defined, let's stablish the concept of *Pinned* and *Non-Pinned* manifests.

When we release a software stack, we test the integration of all the components and validate that they work together to achieve the goal of providing users with a usable code base, thus we tag the manifest and the components to have a reference of this code base validity. This is the *Pinned* manifest and the file name convention is `pinned-<platform>.xml`.

But as software evolves, new features and/or security fixes will be made available in the participating components of the software stack, therefore we provide a manifest where the revision of the components is set to a branch that is updated more frequently with their upstream counterparts. This is the *Non-Pinned* manifest, and the file name simply is the platform name, i.e.: `<platform>.xml`.

CHAPTER

FOUR

TROUBLESHOOTING

The documentation for Neoverse reference design platform software typically suffices in most cases. But there could be certain host development machine dependencies that could cause failures either during build or execution stages. This page provides solutions for known issues that could affect the use of the platform software stack.

4.1 Error while using repo command

The *repo init* or *repo sync* command fails with the below listed error message.

```
File "<path-to-workspace>/.repo/repo/main.py", line 79
file=sys.stderr)
    ^
SyntaxError: invalid syntax
```

The typical reason for this failure could be that the default version of python on the development machine is not python3.6. To resolve this issue, install the latest version of python, if not already installed on the development machine and invoke the repo command from */usr/bin/* with *python3* as listed below.

```
python3 /usr/bin/repo init -u https://git.gitlab.arm.com/infra-solutions/reference-
↪design/infra-refdesign-manifests.git -m pinned-rdv1.xml -b refs/tags/RD-INFRA-2021.02.
↪24
python3 /usr/bin/repo sync -c -j $(nproc) --fetch-submodules --force-sync --no-clone-
↪bundle
```

On systems with python version less than 3.6, there could be further failures as listed below.

```
Traceback (most recent call last):
  File "<path-to-workspace>/.repo/repo/main.py", line 42, in <module>
    from git_config import RepoConfig
  File "<path-to-workspace>/.repo/repo/git_config.py", line 774
    self._Set(f'superproject.{key}', value)
                                   ^
SyntaxError: invalid syntax
```

If *python3* version cannot be updated using the package manager, use the following commands to build and install *python3.7.2* from the source.

```
sudo apt update
sudo apt install build-essential zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev␣
↪libssl-dev libreadline-dev libffi-dev wget libsqlite3-dev python-openssl bzip2
```

(continues on next page)

(continued from previous page)

```
cd /tmp
wget https://www.python.org/ftp/python/3.7.2/Python-3.7.2.tar.xz
tar -xf Python-3.7.2.tar.xz
cd Python-3.7.2
./configure
make -j
sudo make altinstall
```

This will install install *python3.7* in */usr/local/bin/* path and the *repo* command can be invoked using this version.

```
/usr/local/bin/python3.7 /usr/bin/repo init -u https://git.gitlab.arm.com/infra-
↪solutions/reference-design/infra-refdesign-manifests.git -m pinned-rdv1.xml -b refs/
↪tags/RD-INFRA-2021.02.24
/usr/local/bin/python3.7 /usr/bin/repo sync -c -j $(nproc) --fetch-submodules --force-
↪sync --no-clone-bundle
```

4.2 Builds do not progress to completion

During the build of the platform software stack, components such as grub download additional code from remote repositories using the git port (or the git protocol). Development machines on which git port is blocked, the build does not progress to completion, waiting for the additional code to be downloaded. This typically is observed when setting up a new platform software workspace.

As a workaround, use https instead of git protocol for cloning required git submodules of the various components in the software stack. A patch, as an example of this change in the grub component, is listed below.

```
diff --git a/bootstrap b/bootstrap
index 5b08e7e2d..031784582 100755
--- a/bootstrap
+++ b/bootstrap
@@ -47,7 +47,7 @@ PERL="${PERL-perl}"

 me=$0

-default_gnulib_url=git://git.sv.gnu.org/gnulib
+default_gnulib_url=https://git.savannah.gnu.org/git/gnulib.git

 usage() {
   cat <<EOF
```

4.3 FVP closes abruptly

Tests such as distro installation take few hours to complete on Neoverse Reference Design platform FVPs. If the model quits abruptly during its execution without any particular error message displayed in the model launch window, the host machine's memory requirements has to be rechecked. This issue is typically seen if the host machine has a configuration below that of on the one listed at *recommended configuration*.

Repo sync fails when downloading linux repo

If the download of the linux repo fails during the execution of the *repo sync* command, rerun the repo init command with the `--depth=1` parameter appended to the repo init command. The parameter `--depth=1` reduces the commit history that is downloaded and can address this failure in downloading linux repo.

4.4 Error: "/usr/bin/env: 'python': No such file or directory"

repo init could fail if it can't find a compatible reference to python. Please make sure you have the required version of python as mentioned in *install repo* prerequisites section.

If the error still persists, check if */usr/bin* has a binary named python. If you find the binary name to be *python3* (or any *python3.x* for that matter) and /usr/bin/python is not found, then create a softlink to work around this issue as shown below:

```
sudo ln -s /usr/bin/python3 /usr/bin/python
```

CHAPTER

FIVE

REPORT SECURITY VULNERABILITY

Arm Neoverse reference design software solutions are example software projects containing downstream versions of open source components. Although the components in these solutions track their upstream versions, users of these solutions are responsible for ensuring that, if necessary, these components are updated before use to ensure they contain any new functional or security fixes that may be required.

If you think you have found a security vulnerability in a specific open source project which is part of the software stack, it is recommended to follow the vulnerability reporting guidelines specified by the respective project.

If you think you have found a security vulnerability as part of the Neoverse Reference Design platform software stack and does not fall into any specific open source project, then please report by email at arm-security@arm.com specifying the project name as "Neoverse Reference Design Platform Software". More details can be found at Arm Developer website.

CHAPTER

SIX

GETTING STARTED

6.1 Prerequisites

Important:

- Neoverse software stack builds are only supported in linux operating systems.
- The operating system used to validate these instructions is Ubuntu 22.04 (althought any modern linux distribution should work).
- The following sections and chapters assume the commands are executed in a bash shell environment.

Host machine recommended hardware configuration:

- AArch64 or x86-64 architecture host.
- 64GB of free disk space.
- 48GB of RAM (32GB minimum).

The host machine needs the following packages installed.

```
sudo apt update
sudo apt install curl git
```

Configure git as follows.

```
git config --global user.name "<your-name>"
git config --global user.email "<your-email@example.com>"
```

Install repo tool via *'manual method'*. Refer to repo install official documentation as this might change. Instructions provided here for convinience.

```
export REPO=$(mktemp /tmp/repo.XXXXXXXXX)
curl -o ${REPO} https://storage.googleapis.com/git-repo-downloads/repo
gpg --recv-keys 8BB9AD793E8E6153AF0F9A4416530D5E920F5C65
curl -s https://storage.googleapis.com/git-repo-downloads/repo.asc | gpg --verify - $
↪{REPO} && install -m 755 ${REPO} ~/bin/repo
```

Warning: The repo tool requires at least Python 3.6 to be installed on the development machine. On machines where python3 is not the default, the repo init command will fail to complete. Refer the *troubleshooting guide*.

6.2 Download Sources

In the previous section the host machine is configured with the minimum set of tools to allow the user to prepare and *sync* a workspace. This workspace will then configure a build environment, but more on that in the next section.

This workspace is a folder in the user host machine that contains all of the software sources, as well as, build products once a build is successful and complete.

This guide refers to this folder as <workspace> but the user is encouraged to provide a meaningful name.

Create a folder, and change directory to it.

```
mkdir <workspace>
cd <workspace>
```

Initialise and sync (download) the sources. The command below is the generic form and requires <manifest-file-name> and <RELEASE_TAG> to be replaced by valid arguments.

- Manifest file names can be found here.
- Release tags are located in *Release Tags* section of each supported platform user guide or from the release notes.

```
repo init -u https://git.gitlab.arm.com/infra-solutions/reference-design/infra-refdesign-
↪manifests.git -m <manifest-file-name> -b refs/tags/<RELEASE_TAG> --depth=1
repo sync -c -j $(nproc) --fetch-submodules --force-sync --no-clone-bundle
```

Hint: To reduce the size of the commit history that is downloaded (thus reducing the time taken to download the platform software stack), the repo init command above is append with --depth=1. If the user requires more commit history, the argument can be removed before executing the command.

6.3 Build Environment

There are two methods to build the reference stack - host based and container based. The host based build is the traditional one in which a script is executed to install all the build dependencies on the host machine. The container based build is an another method in which a container image is built from a container configuration file and has all the build dependencies satisfied and isolated from the host machine.

Both of the methods assume the user has completed the section *Download Sources*.

6.3.1 Host Based

For setting up the build environment in this method, execute the following command before building the software stack. The execution of this script installs all the build dependencies.

Note: This command installs additional packages on the host machine and so the user is expected to have sufficient privileges on the host machine.

```
./build-scripts/rdinfra/install_prerequisites.sh
```

6.3.2 Container Based

Note: The supported container engine is docker.

Warning: Rootless support changes are distruptive and not compatible with old images. Please *build image* again.

The container image is designed to allow a user to have the sources directory (<workspace>) in the host machine and offload the build stage to the container, thus a user is created inside the container with the same username, user-id and user-group as the user on the linux host machine.

This approach allows a user to have the binaries built by the container and use IDE's like ARM DS to execute debug sessions, as paths and permissions are the same wether inside or outside the container.

Install Container Engine

Please refer to docker install instructions as there are several methods available, ensuring you install the following *docker-engine* and optionally the *buildx-plugin*.

After installation is complete, refer to the post-installation steps on how to manage docker as non-root user.

Build Container Image

Warning: Do **not** execute the wrapper script with root permissions. As doing so, interferes with permissions and will lead to errors when building and executing software.

The wrapper script *container.sh* sets the container file and image name by default and this can be changed with options *-f* and *-i* respectively or by editing the file itself. To see all options available, execute the script with the help flag.

```
cd <workspace>/container-scripts
./container.sh -h
```

To build the container image, execute:

```
./container.sh build
```

Run Container Image

Mount the <workspace> directory in the container by using the option **-v** followed by the absolute path to <workspace>. The mount point inside the container is the exact same path as the host system. To run the container image, execute the following:

```
./container.sh -v /absolute/path/to/rd-infra run
```

The container shall be running and the shell prompt display:

```
$USER:$HOSTNAME:<path_to_rd-infra>$
```

As this is designed to have the same user and hostname as the host, it is not straightforward to see the container is executing, but a way to verify it is to check **.dockerenv** file, thus execute:

```
$ls -la /.dockerenv
```

Rootless Docker Support

Rootless docker support has been added to **container.sh** wrapper script. It will check the current docker context and if rootless is activated it will run containers as rootless containers.

This completes the procedure to setup the container-based build environment.

6.4 Enable Network for FVP's (optional)

If networking is required, the platform FVP's support a virtual ethernet interface that can be configured via TAP mode interface. This mode allows the FVP to be directly connected to the network via a bridge. All ports are forwarded to the FVP networking interface as if it was connected to the network.

6.4.1 Host Dependencies

Note: This command installs additional packages on the host machine and so the user is expected to have sufficient privileges on the host machine.

```
sudo apt update
sudo apt install qemu-kvm libvirt-daemon-system iproute2
```

6.4.2 Configure TAP Interface

Ensure that the libvirtd service is running

```
sudo systemctl start libvirtd
```

Create a network bridge and change state to up. This step is only required once, so the user can skip if a bridge exists.

This example uses `virbr0` for the bridge name.

```
sudo ip link add name virbr0 type bridge
sudo ip link set dev virbr0 up
```

Finally, the TAP interface is created, configured and attached to `virbr0`.

```
sudo ip tuntap add dev tap0 mode tap user $(whoami)
sudo ip link set tap0 promisc on
sudo ip addr add 0.0.0.0 dev tap0
sudo ip link set tap0 up
sudo ip link set tap0 master virbr0
```

This completes the environment setup to have a working workspace so the user can proceed to build, and experiment with Neoverse reference designs features.

Refer the *Troubleshooting Section* for solutions to known issues that might arise during use of the platform software stack.

CHAPTER

SEVEN

LEARNING PATHS

Arm Learning Paths, available on the Developer Hub, are community-created how-to articles about software development for the Arm architecture. They offer detailed tutorials designed to help developers create quality Arm software faster. A Learning Path is a concise tutorial with detailed steps on how to complete a specific task.

The Learning Paths are segmented into categories, each covering different kinds of computer hardware. These categories include Smartphones and Mobile, Laptops and Desktops, Servers and Cloud Computing, Embedded Systems, and Microcontrollers. Neoverse Reference Designs are a part of servers and cloud computing.

The sections below list the available learning paths that are applicable to Neoverse Reference Designs.

7.1 Get started with the Neoverse Reference Design software stack

Follow the link below for this tutorial:

https://learn.arm.com/learning-paths/servers-and-cloud-computing/refinfra-quick-start/

7.2 Debug Neoverse N2 Reference Design with Arm Development Studio

Follow the link below for this tutorial:

https://learn.arm.com/learning-paths/servers-and-cloud-computing/refinfra-debug/

CHAPTER

EIGHT

RD-V3-R1-CFG1 PLATFORM

8.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP.

RD-V3-R1-Cfg1 is a quad-chip platform in which four identical chips are connected through high speed CCG link. The CCG link is enabled through CMN-Cyprus Coherent Multichip Link (CML) feature. RD-V3-R1-Cfg1 platform (a variant of the *RD-V3-R1* platform) also supports the Realm Management Extension (RME). The RD-V3-R1-Cfg1 platform in particular has the following hardware configuration on each chip.

- Up to 8xMP1 Neoverse Poseidon-V3 cores with Direct Connect and 2MB of dedicated, private L2 cache for each core.
- 2 Shared LCP Groups, 4 AP cores per Shared LCP Group.
- CMN S3 Revision 2 interconnect with 3x4 mesh network.
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M55 processor for Runtime Security Engine (RSE) to support Hardware Enforced Security (HES)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)
- Arm Cortex-M55 processor for Local Control Processor (LCP) for local power management of each Application Processor (AP)

The Fixed Virtual Platform of RD-V3-R1-Cfg1 config supports quad chip with 8xMP1 Neoverse Poseidon-V3 CPUs per chip.

The components integrated into this stack are described in *Software Stack* section.

8.2 Platform Specific Details

The following documents provide specific details applicable for RD-V3-R1-Cfg1 Platform:

- Boot Flow
- CMN Cyprus Driver Module
- CMN Cyprus Multichip Configuration
- Image Loading via MCUboot
- Local Control Processor
- Multichip Memory Map

- NI-Tower System Control
- Realm Management Extension
- Runtime Security Engine
- SCP Address Translation Unit Configuration
- SCP - RSE Communication

8.3 Supported Features

RD-V3-R1-Cfg1 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

8.4 Obtaining FVP

The latest version of the RD-V3-R1-Cfg1 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V3 r1 Reference Design FVP" link to obtain the list of available Neoverse RD-V3-R1-Cfg1 Reference Design FVPs.
- Select a FVP build under the section "Download RD-V3-R1-Cfg1" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V3-R1-Cfg1 FVP executable is included in the downloaded installer and named as "FVP_RD_V3_R1". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

8.5 Release Tags

Table below lists the release tags for the RD-V3-R1-Cfg1 platform software stack and the corresponding RD-V3-R1-Cfg1 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V3-R1-Cfg1 FVP Version
RD-INFRA-2025.02.04	11.27.51
RD-INFRA-2024.12.20	11.27.51
RD-INFRA-2024.09.30	11.27.25

CHAPTER

NINE

RD-V3-R1 PLATFORM

9.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP.

RD-V3-R1 is a dual-chip platform in which two identical chips are connected through high speed CCG link. The CCG link is enabled through CMN S3 Coherent Multichip Link (CML) feature. RD-V3-R1 platform also supports the Realm Management Extension (RME). The RD-V3-R1 platform in particular has the following hardware configuration on each chip.

- Up to 70xMP1 Neoverse Poseidon-V3 cores with Direct Connect and 2MB of dedicated, private L2 cache for each core.
- 7 Shared LCP Groups, 10 AP cores per Shared LCP Group.
- CMN S3 Revision 2 (CMN S3 R2) interconnect with 9x8 mesh network.
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M55 processor for Runtime Security Engine (RSE) to support Hardware Enforced Security (HES)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)
- Arm Cortex-M55 processor for Local Control Processor (LCP) for local power management of each Application Processor (AP)

The Fixed Virtual Platform of RD-V3-R1 config supports dual chip with 14xMP1 Neoverse Poseidon-V3 CPUs per chip (2 AP cores per LCP Group)

The components integrated into this stack are described in *Software Stack* section.

9.2 Platform Specific Details

The following documents provide specific details applicable for RD-V3-R1 Platform:

- Boot Flow
- CMN Cyprus Driver Module
- CMN Cyprus Multichip Configuration
- Image Loading via MCUboot
- Local Control Processor
- Multichip Memory Map

- NI-Tower System Control
- Realm Management Extension
- Runtime Security Engine
- SCP Address Translation Unit Configuration
- SCP - RSE Communication

9.3 Supported Features

RD-V3-R1 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

9.4 Obtaining FVP

The latest version of the RD-V3-R1 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V3 r1 Reference Design FVP" link to obtain the list of available Neoverse RD-V3-R1 Reference Design FVPs.
- Select a FVP build under the section "Download RD-V3-R1" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V3-R1 FVP executable is included in the downloaded installer and named as "FVP_RD_V3_R1". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

9.5 Release Tags

Table below lists the release tags for the RD-V3-R1 platform software stack and the corresponding RD-V3-R1 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V3-R1 FVP Version
RD-INFRA-2025.02.04	11.27.51
RD-INFRA-2024.12.20	11.27.51
RD-INFRA-2024.09.30	11.27.25

CHAPTER

TEN

RD-V3 CFG2 PLATFORM

10.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-V3-Cfg2 platform is a quad chip variant of the *RD-V3* platform.

RD-V3-Cfg2 is a quad-chip platform in which four identical chips are connected through high speed CCG link. The CCG link is enabled through CMN-Cyprus Coherent Multichip Link (CML) feature. The RD-V3-Cfg2 platform in particular has the following hardware configuration on each chip.

- 4x32XMP1 Neoverse Poseidon-V cores with Direct Connect and 2MB of dedicated, private L2 cache for each core.
- CMN-Cyprus interconnect with 7x6 mesh network.
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M55 processor for Runtime Security Engine (RSE) to support Hardware Enforced Security (HES)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)
- Arm Cortex-M55 processor for Local Control Processor (LCP) for local power management of each Application Processor (AP)

The Fixed Virtual Platform of RD-V3-Cfg2 config supports quad chip with 4xMP1 Neoverse Poseidon-V CPUs per chip.

The components integrated into this stack are described in *Software Stack* section.

10.2 Platform Specific Details

The following documents provide specific details applicable for RD-V3-Cfg2 Platform:

- Boot Flow
- CMN Cyprus Driver Module
- CMN Cyprus Multichip Configuration
- Image Loading via MCUboot
- Local Control Processor
- Multichip Memory Map
- NI-Tower System Control

- Realm Management Extension
- Runtime Security Engine
- SCP Address Translation Unit Configuration
- SCP - RSE Communication

10.3 Supported Features

RD-V3-Cfg2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Low power idle*
- *Collaborative processor performance control*
- *AP Reset to BL31*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

10.4 Obtaining FVP

The latest version of the RD-V3-Cfg2 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V3 Reference Design FVP" link to obtain the list of available Neoverse RD-V3 Reference Design FVPs.
- Select a FVP build under the section "Download RD-V3" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V3-Cfg2 FVP executable is included in the downloaded installer and named as "FVP_RD_V3_Cfg2". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

10.5 Release Tags

Table below lists the release tags for the RD-V3-Cfg2 platform software stack and the corresponding RD-V3-Cfg2 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V3-Cfg2 FVP Version
RD-INFRA-2025.02.04	11.27.51
RD-INFRA-2024.12.20	11.27.51
RD-INFRA-2024.09.30	11.27.25
RD-INFRA-2024.07.15	11.26.15
RD-INFRA-2024.04.17	11.24.16
RD-INFRA-2024.01.16	11.24.16
RD-INFRA-2023.09.28	11.23.11
RD-INFRA-2023.06.28	11.22.16

CHAPTER

ELEVEN

RD-V3 CFG1 PLATFORM

11.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-V3-Cfg1 platform (a variant of the *RD-V3* platform) also supports the Realm Management Extension (RME) and is based on the following hardware configuration.

- 8xMP1 Neoverse Poseidon-V cores with Direct Connect and 2MB of dedicated, private L2 cache for each core.
- CMN-Cyprus interconnect with 3x3 mesh network.
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M55 processor for Runtime Security Engine (RSE) to support Hardware Enforced Security (HES)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)
- Arm Cortex-M55 processor for Local Control Processor (LCP) for local power management of each Application Processor (AP)

The components integrated into this stack are described in *Software Stack* section.

11.2 Platform Specific Details

The following documents provide specific details applicable for RD-V3-Cfg1 Platform:

- AP - RSE Attestation Service
- Boot Flow
- CMN Cyprus Driver Module
- Image Loading via MCUboot
- Local Control Processor
- NI-Tower Network-on-Chip Interconnect
- NI-Tower System Control
- Realm Management Extension
- Runtime Security Engine
- SCP Address Translation Unit Configuration
- SCP - RSE Communication

11.3 Supported Features

RD-V3-Cfg1 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *UEFI Secure Boot*
- *Reliability, Availability, and Serviceability*
  - *Poseidon CPU/RAM Error Injection Tests*
  - *CMN Cyprus Kernel First Handling*
  - *SCP Error Injection Utility*
- *MPAM-resctrl*
- *Low power idle*
- *Collaborative processor performance control*
- *AP Reset to BL31*

11.4 Obtaining FVP

The latest version of the RD-V3-Cfg1 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V3 Reference Design FVP" link to obtain the list of available Neoverse RD-V3-Cfg1 Reference Design FVPs.
- Select a FVP build under the section "Download RD-V3-Cfg1" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V3-Cfg1 FVP executable is included in the downloaded installer and named as "FVP_RD_V3_Cfg1". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

11.5 Release Tags

Table below lists the release tags for the RD-V3-Cfg1 platform software stack and the corresponding RD-V3-Cfg1 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V3-Cfg1 FVP Version
RD-INFRA-2025.02.04	11.27.51
RD-INFRA-2024.12.20	11.27.51
RD-INFRA-2024.09.30	11.27.25
RD-INFRA-2024.07.15	11.26.15
RD-INFRA-2024.04.17	11.24.16
RD-INFRA-2024.01.16	11.24.16
RD-INFRA-2023.09.28	11.23.11
RD-INFRA-2023.06.28	11.22.16
RD-INFRA-2023.03.29	11.21.18

CHAPTER

TWELVE

RD-V3 PLATFORM

12.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-V3 is the first RD platform with Realm Management Extension (RME) support and is based on the following hardware configuration.

- 32xMP1 Neoverse Poseidon-V cores with Direct Connect and 2MB of dedicated, private L2 cache for each core.
- CMN-Cyprus interconnect with 7x6 mesh network.
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M55 processor for Runtime Security Engine (RSE) to support Hardware Enforced Security (HES)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)
- Arm Cortex-M55 processor for Local Control Processor (LCP) for local power management of each Application Processor (AP)

The Fixed Virtual Platform of RD-V3 config supports 16xMP1 Neoverse Poseidon-V CPUs.

The components integrated into this stack are described in *Software Stack* section.

12.2 Platform Specific Details

The following documents provide specific details applicable for RD-V3 Platform:

- AP - RSE Attestation Service
- Boot Flow
- Chain of Trust (CoT) for CCA
- CMN Cyprus Driver Module
- Image Loading via MCUboot
- Local Control Processor
- NI-Tower Network-on-Chip Interconnect
- NI-Tower System Control
- Realm Management Extension
- Runtime Security Engine
- SCP Address Translation Unit Configuration

- SCP - RSE Communication

12.3 Supported Features

RD-V3 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Arm SystemReady Compliance*
- *UEFI Secure Boot*
- *Low power idle*
- *Collaborative processor performance control*
- *Reboot-Shutdown test*
- *RD-V3 SMCF test*
- *AP Reset to BL31*
- *Virtualization*
  - *IO virtualization*
  - *Virtual Interrupts And VGIC*
  - *KVM Unit Test*
  - *Booting Distro as a VM*

12.4 Obtaining FVP

The latest version of the RD-V3 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V3 Reference Design FVP" link to obtain the list of available Neoverse RD-V3 Reference Design FVPs.
- Select a FVP build under the section "Download RD-V3" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V3 FVP executable is included in the downloaded installer and named as "FVP_RD_V3". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

12.5 Release Tags

Table below lists the release tags for the RD-V3 platform software stack and the corresponding RD-V3 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V3 FVP Version
RD-INFRA-2025.02.04	11.27.51
RD-INFRA-2024.12.20	11.27.51
RD-INFRA-2024.09.30	11.27.25
RD-INFRA-2024.07.15	11.26.15
RD-INFRA-2024.04.17	11.24.16
RD-INFRA-2024.01.16	11.24.16
RD-INFRA-2023.09.28	11.23.11
RD-INFRA-2023.06.28	11.22.16
RD-INFRA-2023.03.29	11.21.18

CHAPTER

THIRTEEN

RD-V2 PLATFORM

13.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-V2 in particular is based on the following hardware configuration.

- 32xMP1 Neoverse V2 CPUs
- CMN-700 interconnect (mesh size 6x6)
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The Fixed Virtual Platform of RD-V2 config supports 16xMP1 Neoverse V2 CPUs.

The components integrated into this stack are described in *Software Stack* section.

13.2 Supported Features

RD-V2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Arm SystemReady Compliance*
- *UEFI Secure Boot*
- *Virtualization* [1]
  - *IO virtualization*
  - *Virtual Interrupts And VGIC*
  - *KVM Unit Test*
  - *Booting Distro as a VM*
- *Non-discoverable IO Virtualization block*
- *Trusted Firmware-A Tests* [1]
- *Virtio-P9*

- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

[1] Build and boot not supported on AArch64 host machines.

13.3 Obtaining FVP

The latest version of the RD-V2 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse V2 Reference Design FVP" link to obtain the list of available Neoverse RD-V2 Reference Design FVPs.
- Select a FVP build based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-V2 FVP executable is included in the downloaded installer and named as "FVP_RD_V2". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

13.4 Release Tags

Table below lists the release tags for the RD-V2 platform software stack and the corresponding RD-V2 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V2 FVP Version
RD-INFRA-2024.12.20	11.24.12
RD-INFRA-2024.09.30	11.24.12
RD-INFRA-2024.07.15	11.24.12
RD-INFRA-2024.04.17	11.24.12
RD-INFRA-2023.12.22	11.24.12
RD-INFRA-2023.09.29	11.20.18
RD-INFRA-2023.06.30	11.20.18
RD-INFRA-2023.03.31	11.20.18

CHAPTER

FOURTEEN

RD-N2 CFG3 PLATFORM

14.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N2 Cfg3 platform (which is a variant of the the *RD-N2* platform) in particular is based on the following hardware configuration.

- 16xMP1 Neoverse N2 CPUs
- CMN-700 interconnect (mesh size 10x6)
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The components integrated into this stack are described in *Software Stack* section.

14.2 Supported Features

RD-N2 Cfg3 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Arm SystemReady Compliance*
- *UEFI Secure Boot*
- *Virtualization* [1]
  - *IO virtualization*
  - *Virtual Interrupts And VGIC*
  - *KVM Unit Test*
  - *Booting Distro as a VM*
- *Non-discoverable IO Virtualization block*
- *Virtio-P9*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

[1] Build and boot supported on AArch64 host machines as well.

14.3 Obtaining FVP

The latest version of the RD-N2 Cfg3 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse N2 Reference Design FVP" link to obtain the list of available Neoverse RD-N2 Reference Design FVPs.
- Select a FVP build under the section "Download RD-N2 Cfg3" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-N2 Cfg3 FVP executable is included in the downloaded installer and named as "FVP_RD_N2_Cfg3". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

14.4 Release Tags

Table below lists the release tags for the RD-N2 Cfg3 platform software stack and the corresponding RD-N2 Cfg3 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N2 Cfg3 FVP Version
RD-INFRA-2024.12.20	11.25.23
RD-INFRA-2024.09.30	11.25.23
RD-INFRA-2024.07.15	11.25.23
RD-INFRA-2024.04.17	11.25.23
RD-INFRA-2023.12.22	11.24.12
RD-INFRA-2023.09.29	11.20.18
RD-INFRA-2023.06.30	11.20.18
RD-INFRA-2023.03.31	11.20.18

CHAPTER

FIFTEEN

RD-N2 CFG2 PLATFORM

15.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N2 Cfg2 platform (which is a variant of the the *RD-N2* platform) is a quad-chip platform in which four identical chips are connected through high speed CCG link. The CCG link is enabled through CMN-700 Coherent Multichip Link (CML) feature. RD-N2 Cfg2 in particular has the following hardware configuration on each chip.

- 4xMP1 Neoverse N2 CPUs
- CMN-700 interconnect (mesh size 6x6)
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The components integrated into this stack are described in *Software Stack* section.

15.2 Supported Features

RD-N2 Cfg2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

15.3 Obtaining FVP

The latest version of the RD-N2 Cfg2 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse N2 Reference Design FVP" link to obtain the list of available Neoverse RD-N2 Reference Design FVPs.
- Select a FVP build under the section "Download RD-N2" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-N2 Cfg2 FVP executable is included in the downloaded installer and named as "FVP_RD_N2_Cfg2". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

15.4 Release Tags

Table below lists the release tags for the RD-N2 Cfg2 platform software stack and the corresponding RD-N2 Cfg2 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N2 Cfg2 FVP Version
RD-INFRA-2024.12.20	11.25.23
RD-INFRA-2024.09.30	11.25.23
RD-INFRA-2024.07.15	11.25.23
RD-INFRA-2024.04.17	11.25.23
RD-INFRA-2023.12.22	11.24.12
RD-INFRA-2023.09.29	11.20.18
RD-INFRA-2023.06.30	11.20.18
RD-INFRA-2023.03.31	11.20.18

CHAPTER

SIXTEEN

RD-N2 CFG1 PLATFORM

16.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N2 Cfg1 platform (which is a variant of the the *RD-N2* platform) in particular is based on the following hardware configuration.

- 8xMP1 Neoverse N2 CPUs
- CMN-700 interconnect (mesh size 3x3)
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The components integrated into this stack are described in *Software Stack* section.

16.2 Supported Features

RD-N2 Cfg1 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *UEFI Secure Boot*
- *Non-discoverable IO Virtualization block*
- *Virtualization* [1]
  - *IO virtualization*
  - *Virtual Interrupts And VGIC*
  - *KVM Unit Test*
  - *Booting Distro as a VM*
- *Reliability, Availability, and Serviceability*
  - *RD-N2 CPU/RAM Error Injection Tests*
- *Trusted Firmware-A Tests* [1]
- *Virtio-P9*

- *MPAM-resctrl*
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

[1] Build and boot not supported on AArch64 host machines.

16.3 Obtaining FVP

The latest version of the RD-N2 Cfg1 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse N2 Reference Design FVP" link to obtain the list of available Neoverse RD-N2 Reference Design FVPs.
- Select a FVP build under the section "Download RD-N2 Cfg1" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-N2 Cfg1 FVP executable is included in the downloaded installer and named as "FVP_RD_N2_Cfg1". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

16.4 Release Tags

Table below lists the release tags for the RD-N2 Cfg1 platform software stack and the corresponding RD-N2 Cfg1 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N2 Cfg1 FVP Version
RD-INFRA-2024.12.20	11.25.23
RD-INFRA-2024.09.30	11.25.23
RD-INFRA-2024.07.15	11.25.23
RD-INFRA-2024.04.17	11.25.23
RD-INFRA-2023.12.22	11.24.12
RD-INFRA-2023.09.29	11.20.18
RD-INFRA-2023.06.30	11.20.18
RD-INFRA-2023.03.31	11.20.18

CHAPTER

SEVENTEEN

RD-N2 PLATFORM

17.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N2 in particular is based on the following hardware configuration.

- 32xMP1 Neoverse N2 CPUs
- CMN-700 interconnect
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The Fixed Virtual Platform of RD-N2 config supports 16xMP1 Neoverse N2 CPUs.

The components integrated into this stack are described in *Software Stack* section.

17.2 Supported Features

RD-N2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *Arm SystemReady Compliance*
- *UEFI Secure Boot*
- *Virtualization* [1]
  - *IO virtualization*
  - *Virtual Interrupts And VGIC*
  - *KVM Unit Test*
  - *Booting Distro as a VM*
- *Non-discoverable IO Virtualization block*
- *Trusted Firmware-A Tests* [1]
- *Virtio-P9*

- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

[1] Build and boot not supported on AArch64 host machines.

17.3 Obtaining FVP

The latest version of the RD-N2 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page,

- Navigate to "Neoverse Infrastructure FVPs" section.
- Click on "Neoverse N2 Reference Design FVP" link to obtain the list of available Neoverse RD-N2 Reference Design FVPs.
- Select a FVP build under the section "Download RD-N2" based on the host machine architecture.
  - For AArch64 host machine, click "Download Linux - Arm Host (DEV)" link.
  - For x86-64 host machine, click "Download Linux" link.

The RD-N2 FVP executable is included in the downloaded installer and named as "FVP_RD_N2". Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

17.4 Release Tags

Table below lists the release tags for the RD-N2 platform software stack and the corresponding RD-N2 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N2 FVP Version
RD-INFRA-2024.12.20	11.25.23
RD-INFRA-2024.09.30	11.25.23
RD-INFRA-2024.07.15	11.25.23
RD-INFRA-2024.04.17	11.25.23
RD-INFRA-2023.12.22	11.24.12
RD-INFRA-2023.09.29	11.20.18
RD-INFRA-2023.06.30	11.20.18
RD-INFRA-2023.03.31	11.20.18

CHAPTER

EIGHTEEN

RD-V1 MC PLATFORM

18.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP.

RD-V1 MC is a quad-chip platform in which four identical chips are connected through high speed CCIX link. The CCIX link is enabled through CMN-650 Coherent Multichip Link (CML) feature. RD-V1 MC in particular is based on the following hardware configuration.

- 128xMP1 Neoverse V1 CPUs
- CMN-650 interconnect
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The Fixed Virtual Platform of RD-V1 MC supports a reduced configuration of the above configuration. That is, there are a total of 16xMP1 Neoverse V1 CPUs, 4xMP1 Neoverse CPUs on each chip on RD-V1 MC FVP.

The components integrated into this stack are described in *Software Stack* section.

18.2 Supported Features

RD-V1 MC platform software stack supports the following features.

- *Busybox Boot*.
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

18.3 Obtaining FVP

The latest version of the RD-V1 MC fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page, navigate to "Neoverse Infrastructure FVPs" section to download the RD-V1 platform FVP installer.

Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

18.4 Release Tags

Table below lists the release tags for the RD-V1 Quad-Chip platform software stack and the corresponding RD-V1 MC FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V1 MC FVP Version
RD-INFRA-2024.12.20	11.17.29
RD-INFRA-2024.09.30	11.17.29
RD-INFRA-2024.07.15	11.17.29
RD-INFRA-2024.04.17	11.17.29
RD-INFRA-2023.12.22	11.17.29
RD-INFRA-2023.09.29	11.17.29
RD-INFRA-2023.06.30	11.17.29
RD-INFRA-2023.03.31	11.17.29

CHAPTER

NINETEEN

RD-V1 PLATFORM

19.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-V1 in particular is based on the following hardware configuration.

- 32xMP1 Neoverse V1 CPUs
- CMN-650 interconnect
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The Fixed Virtual Platform of RD-V1 supports 16xMP1 Neoverse V1 CPUs.

The components integrated into this stack are described in *Software Stack* section.

19.2 Supported Features

RD-V1 platform software stack supports the following features.

- *Busybox Boot*
- *Linux Distribution Boot*
- *Low power idle*
- *Collaborative processor performance control*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

19.3 Obtaining FVP

The latest version of the RD-V1 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page, navigate to "Neoverse Infrastructure FVPs" section to download the RD-V1 platform FVP installer.

Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

19.4 Release Tags

Table below lists the release tags for the RD-V1 platform software stack and the corresponding RD-V1 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-V1 FVP Version
RD-INFRA-2024.12.20	11.17.29
RD-INFRA-2024.09.30	11.17.29
RD-INFRA-2024.07.15	11.17.29
RD-INFRA-2024.04.17	11.17.29
RD-INFRA-2023.12.22	11.17.29
RD-INFRA-2023.09.29	11.17.29
RD-INFRA-2023.06.30	11.17.29
RD-INFRA-2023.03.31	11.17.29

CHAPTER

TWENTY

RD-N1 EDGE X2 PLATFORM

20.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N1 Edge in particular is based on the following hardware configuration.

- 8x Neoverse N1 Cores with DynamIQ Shared Unit (DSU)
- Dedicated L2 cache: 512KB per core
- Shared L3 cache: 2MB per cluster
- CMN-600 with CML option: 8MB System Level Cache and 16MB Snoop Filter
- DMC-620 with 2xRDIMM DDR4-3200
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

RD-N1 Edge dual-chip is a platform configuration in which two RD-N1 Edge platforms are connected through high speed CCIX link. The CCIX link is enabled by CMN600's Coherent Multichip Link. Such platforms are called RD-N1 Edge-Dual hereafter.

The components integrated into this stack are described in *Software Stack* section.

20.2 Supported Features

RD-N1 Edge-Dual platform software stack supports the following features.

- *Busybox Boot*.
- *Low power idle*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

20.3 Obtaining FVP

The latest version of the RD-N1 Edge-Dual fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page, navigate to "Neoverse Infrastructure FVPs" section to download the RD-N1 Edge platform FVP installer.

Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

20.4 Release Tags

Table below lists the release tags for the RD-N1 Edge dual-chip platform software stack and the corresponding RD-N1 Edge dual-chip FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N1 Edge-Dual FVP Version
RD-INFRA-2024.12.20	11.17.29
RD-INFRA-2024.09.30	11.17.29
RD-INFRA-2024.07.15	11.17.29
RD-INFRA-2024.04.17	11.17.29
RD-INFRA-2023.12.22	11.17.29
RD-INFRA-2023.09.29	11.17.29
RD-INFRA-2023.06.30	11.17.29
RD-INFRA-2023.03.31	11.17.29

CHAPTER

TWENTYONE

RD-N1 EDGE PLATFORM

21.1 Overview

RD (Reference Design) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. RD-N1 Edge in particular is based on the following hardware configuration.

- 8x Neoverse N1 Cores with DynamIQ Shared Unit (DSU)
- Dedicated L2 cache: 512KB per core
- Shared L3 cache: 2MB per cluster
- CMN-600 with CML option: 8MB System Level Cache and 16MB Snoop Filter
- DMC-620 with 2xRDIMM DDR4-3200
- Multiple AXI expansion ports for I/O Coherent PCIe, Ethernet, offload
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The components integrated into this stack are described in *Software Stack* section.

21.2 Supported Features

RD-N1 Edge platform software stack supports the following features.

- *Busybox Boot*.
- *Linux Distribution Boot*
- *Low power idle*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

21.3 Obtaining FVP

The latest version of the RD-N1 Edge fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page, navigate to "Neoverse Infrastructure FVPs" section to download the RD-N1 Edge platform FVP installer.

Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

21.4 Release Tags

Table below lists the release tags for the RD-N1 Edge platform software stack and the corresponding RD-N1 Edge FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	RD-N1 Edge FVP Version
RD-INFRA-2024.12.20	11.17.29
RD-INFRA-2024.09.30	11.17.29
RD-INFRA-2024.07.15	11.17.29
RD-INFRA-2024.04.17	11.17.29
RD-INFRA-2023.12.22	11.17.29
RD-INFRA-2023.09.29	11.17.29
RD-INFRA-2023.06.30	11.17.29
RD-INFRA-2023.03.31	11.17.29

CHAPTER

TWENTYTWO

SGI-575 PLATFORM

22.1 Overview

SGI (System Guidance for Infrastructure) is a collection of resources to provide a representative view of typical compute subsystems that can be designed and implemented using specific generations of Arm IP. SGI-575 in particular is based on the following hardware configuration.

- 8x Cortex-A75 with private L2 Cache
- DynamIQ with L3 Cache options
- System Level Cache options
- Up to 2x DDR4-3200 (DMC-620)
- Arm Cortex-M7 for System Control Processor (SCP) and Manageability Control Processor (MCP)

The components integrated into this stack are described in *Software Stack* section.

22.2 Supported Features

SGI-575 platform software stack supports the following features.

- *Busybox Boot*
- *Low power idle*

Follow the links above for detailed information about the build and execute steps for each of the supported features.

22.3 Obtaining the FVP

The latest version of the SGI-575 fixed virtual platform (FVP) can be downloaded from the Arm Ecosystem FVPs page. On this page, navigate to "Neoverse Infrastructure FVPs" section to download the SGI-575 platform FVP installer.

Follow the instructions of the installer and setup the FVP. The installer, by default, selects the home directory to install the FVP. To opt for different directory than the one selected by the installer, provide an absolute path to that directory when prompted for during the FVP installation process.

22.4 Release Tags

Table below lists the release tags for the SGI-575 platform software stack and the corresponding SGI-575 FVP version that is recommended to be used along with the listed release tag. The summary of the changes introduced and tests validated in each release is listed in the release note, the link to which is in the 'Release Tag' column in the table below.

Release Tag	SGI-575 FVP Version
RD-INFRA-2024.09.30	11.15.26
RD-INFRA-2024.07.15	11.15.26
RD-INFRA-2024.04.17	11.15.26
RD-INFRA-2023.12.22	11.15.26
RD-INFRA-2023.09.29	11.15.26
RD-INFRA-2023.06.30	11.15.26
RD-INFRA-2023.03.31	11.15.26

CHAPTER

TWENTYTHREE

AP BOOT FROM BL31 (RESET TO BL31 FLOW)

Important: This feature might not be applicable to all platforms. Please check individual platform pages, section **Supported Features** to confirm if this feature is listed as supported.

23.1 Overview of Reset to BL31

Trusted Firmware-A (TF-A) has three bootloader stages called BL1, BL2 and BL31. BL1 and BL2 serve the purpose of platform initialization and loading other firmware images (including BL31). BL31 is the runtime resident. TF-A has a feature called RESET_TO_BL31 which allows the application processor to reset directly to the BL31 stage, bypassing BL1 and BL2.

23.2 Building the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Reset to BL31 is disabled for all platforms by default. To build the platform software for Reset to BL31 boot flow, an environment variable $PLAT_RESET_AP_TO_BL31 needs to be set to 1. If this variable is not defined or is set to 0, Reset to BL31 remains disabled. Any values other than 0 and 1 will result in a build failure.

When $PLAT_RESET_AP_TO_BL31 is 1, all the AP binaries that are needed for the setup of subsequent images are preloaded in the model.

For example, to build the software for Busybox boot, the commands are as follows:

```
export PLAT_RESET_AP_TO_BL31=1
./build-scripts/rdinfra/build-test-busybox.sh -p <platform name> <command>
```

Supported command line options are listed below.

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - `clean`
  - `build`

- `package`
- `all` (all of the three above)

23.3 Booting platforms with Reset to BL31 boot flow

When platform software is built with $PLAT_RESET_AP_TO_BL31 set to 1, a file called boot_info.sh is created in the platform's output directory. This file contains the necessary information for BL31-based boot. No additional parameters are needed to be passed for booting. The boot commands remain the same as those specified under *Boot Operating System(s)*

CHAPTER

TWENTYFOUR

BOOT OPERATING SYSTEM(S)

24.1 Busybox Boot

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Busybox is a lightweight executable which packages lots of POSIX compliant UNIX utilities in a single file system. Busybox boot with Neoverse Reference Design (RD) platform software stack demonstrates the integration of various software components on the software stack resulting in the ability to boot linux kernel on RD fixed virtual platform (FVP).

Booting to busybox is especially helpful when porting the software stack for new platforms which are derivative of Neoverse reference design platform as this can be quickly executed to ensure that the various software components are properly integrated and verify the basic functionality of various software components.

This document describes how to build the Neoverse RD platform software stack and and use it to boot upto busybox on the Neoverse RD FVP.

24.1.1 Build the platform software

This section describes the procedure to build the disk image for busybox boot. The disk image consists of two partitions. The first partition is a EFI partition and contains grub. The second partition is a ext3 partition which contains the linux kernel image. Examples on how to use the build command for busybox boot are listed below.

To build the software stack, the command to be used is

```
./build-scripts/rdinfra/build-test-busybox.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`

* build
* package
* all (all of the three above)

Note: On networks where git port is blocked, the build procedure might not progress. Refer the *troubleshooting guide* for possible ways to resolve this issue.

Examples of the build command are

- Command to clean, build and package the RD-N2 software stack required for busybox boot on RD-N2 platform:

```
./build-scripts/rdinfra/build-test-busybox.sh -p rdn2 all
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform.

```
./build-scripts/rdinfra/build-test-busybox.sh -p rdn2 build
```

Note: This command should be followed by the package command to complete the preparation of the FIP and the disk image.

- Command to package the previously built software stack and prepare the FIP and the disk image.

```
./build-scripts/rdinfra/build-test-busybox.sh -p rdn2 package
```

24.1.2 Boot upto Busybox

After the build of the platform software stack for busybox boot is complete, the following commands can be used to start the execution of the *selected platform fastmodel* and boot the platform up to the busybox prompt. Examples on how to use the command are listed below.

To boot up to the busybox prompt, the commands to be used are

- Set MODEL path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Launch busybox boot:

```
./boot.sh -p <platform name> -a <additional_params> -n [true|false]
```

Supported command line options are listed below

- -p <platform name>

  - Lookup for a platform name in Platform Names.

- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to boot upto busybox are as listed below.

- Command to start the execution of the RD-N2 model to boot up to the Busybox prompt:

```
./boot.sh -p rdn2
```

- Command to start the execution of the RD-N2 model to boot up to the Busybox prompt with network enabled. The model supports virtio.net allowing the software running within the model to access the network:

```
./boot.sh -p rdn2 -n true
```

- Command to start the execution of the RD-N2 model with networking enabled and to boot up to the Busybox prompt. Additional parameters to the model are supplied using the `-a` command line parameter:

```
./boot.sh -p rdn2 -n true -a "-C board.flash0.diagnostics=1"
```

24.2 Buildroot Boot

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Buildroot is a simple, efficient and easy-to-use tool to generate a complete Linux systems through cross-compilation. In order to achieve this, buildroot is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for a target and can be used for any combination, independently, one can for example use an existing cross- compilation toolchain, and build only the root filesystem with buildroot.

Buildroot supports numerous processors and their variants from various families such as, PowerPC, MIPS, and ARM processors, etc. It comes with default configurations for several boards available off-the-shelf.

Online documentation for buildroot can be found here.

Buildroot boot on Neoverse Reference Design platforms allows the use of buildroot as the filesystem and boot the software stack on the fast model. This document describes the procedure to build and execute the software stack with buildroot as the root filesystem.

24.2.1 Build the platform software

This section describes the procedure to build the disk image for buildroot boot. The disk image consists of two partitions. The first partition is a EFI partition and contains grub. The second partition is a ext3 partition and contains the linux kernel image. Examples on how to use the build command for buildroot boot are listed below.

To build the software stack, the command to be used is

```
./build-scripts/rdinfra/build-test-buildroot.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`
    - `build`
    - `package`
    - `all` (all of the three above)

Examples of the build command are

- Command to clean, build and package the software stack needed for the buildroot boot on RD-N2 platform:

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2 all
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform.

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2 build
```

Note: This command should be followed by the `package` command to complete the preparation of the FIP and the disk image.

- Command to package the previously built software stack and prepares the FIP and the disk image.

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2 package
```

24.2.2 Modifying buildroot target filesystem (optional)

Buildroot supports a number of pre-configured packages, customizations across various components, supports a number of pre-configured packages, and also allows adding or modifying files on the target filesystem. This provides the ability to create a richer filesystem compared to busybox.

Though not recommended, for temporary modifications, it is possible to modify the buildroot target filesystem directly and rebuild the image. The target file- system is available under out/arm64/target/ directory in buildroot source. After making required changes, build the software stack again to rebuild the target filesystem image.

Note: If the buildroot repo is cleaned, these changes will be lost.

After the changes are made, run the build command for buildroot and package it. Examples of the incremental build command are

- Command to perform an incremental build of the buildroot component included in the software stack for the RD-N2 platform.

```
./build-scripts/build-buildroot.sh -p rdn2 build
```

- Command to package the previously built software stack and prepares the FIP and the disk image.

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2 package
```

24.2.3 Booting with Buildroot as the filesystem

After the build of the platform software stack for buildroot boot is complete, the following command starts the execution of the *selected platform fastmodel* and the software boots up to the buildroot prompt. Examples on how to use the command are listed below.

To boot up to the buildroot prompt, the command to be used is

- Set `MODEL` path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Launch buildroot boot:

```
./boot-buildroot.sh -p <platform name> -a <additional_params> -n␣
↪[true|false]
```

Supported command line options are listed below

- -p <platform name>
  - Lookup a platform name in Platform Names.
- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to boot with buildroot as the filesystem are as listed below.

- Command to start the execution of the RD-N2 model to boot up to the buildroot prompt:

```
./boot-buildroot.sh -p rdn2
```

- Command to start the execution of the RD-N2 model to boot up to the buildroot prompt with network enabled. The model supports virtio.net allowing the software running within the model to access the network:

```
./boot-buildroot.sh -p rdn2 -n true
```

- Command to start the execution of the RD-N2 model with networking enabled and to boot up to the buildroot prompt. Additional parameters to the model are supplied using the `-a` command line parameter:

```
./boot-buildroot.sh -p rdn2 -n true -a "-C board.flash0.diagnostics=1"
```

24.3 Distro Boot (and Install)

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Neoverse Reference Design (RD) platform software stack supports boot of various linux distributions such as Debian, Ubuntu or Fedora.

This can be achieved either by using a pre-installed (raw) image of the distribution or performing an installation from an iso image.

The prefered method is to use the pre-installed images as greatly reduces the time needed to validate that the software stack can boot into Linux.

An installation from an iso image can take several hours, and will vary greatly with the hardware used, but even on modern hardware can be around 8 hours.

Regardless of the approach selected, the common step is to build the software stack as mentioned below in *Build the platform software*.

24.3.1 Build the platform software

This section describes the procedure to build the platform firmware required to boot or install a linux distribution on Neoverse RD platforms.

To build the RD software stack, the command to be used is

```
./build-scripts/build-test-uefi.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - `clean`
  - `build`

  - `package`
  - `all` (all of the three above)

Using RD-N2 as an example:

- Command to clean, build and package the software stack:

```
./build-scripts/build-test-uefi.sh -p rdn2 all
```

- Command to remove the generated outputs (binaries):

```
./build-scripts/build-test-uefi.sh -p rdn2 clean
```

Important: If using incremental builds, use target command `build` followed by `package`, so the output binaries are correctly generated.

- Command to perform an incremental build of the software stack:

```
./build-scripts/build-test-uefi.sh -p rdn2 build
./build-scripts/build-test-uefi.sh -p rdn2 package
```

24.3.2 Boot a Linux Distribution

Pre-Installed (Raw) images

Debian Distribution

The cloud images for Debian can be obtained from the Debian cloud images page. A number of images listed by codename, along with the daily builds of the latest release version are available on this page.

Important:

- It is recommended to use the nocloud variant of the Debian cloud image as it provisions the user to login as root without a password.
- Select an image for the `aarch64` architecture. Which can also be named `arm64`.
- Select an image with file extension `.raw`.

As an example, to download the image navigate as follows:

- `bookworm/` >> `latest` >> `debian-12-nocloud-arm64.raw`

Using RD-N2 as an example, set `MODEL` environment variable to the FVP path, and run the boot script with argument `-d` to the downloaded image path.

```
export MODEL=<absolute/path/to/FVP/binary>
cd model-scripts/rdinfra
./distro.sh -p rdn2 -d <absolute/path/to/image>
```

Supported command line options are listed below

- -p <platform name>

  - Lookup for a platform name in Platform Names.
- -d <satadisk_path>
  - Absolute path to the installed distro disk image created using the instructions listed in the previous section.
- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

24.3.3 Install a Linux Distribution

After the build of the platform software stack is complete, a distribution can be installed into a SATA disk image. Before beginning the installation process, download the CD iso image of the required distribution version. See below Linux distributions downloads pages:

- Fedora
- Ubuntu
- Debian

Important: Select an image for the `aarch64` architecture. Which can also be named `arm64`.

The generic command to perform the installation is:

```
./distro.sh -p <platform name> -i <abs_iso_image_path> -s <disk size> -a <additional_
↪params> -n [true|false]
```

Supported command line options are listed below

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -i <abs_iso_image_path>
  - Absolute path to the downloaded distribution installer disk image.
- -s <disk_size>
  - Size of the SATA disk image (in GB) to be created. 12GB and above is good enough for most use cases.
- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

As an example:

```
export MODEL=<absolute/path/to/FVP/binary>
cd model-scripts/rdinfra
./distro.sh -p rdn2 -i <absolute/path/to/iso> -s 16
```

- This command creates a 16GB SATA disk image, boots the selected platform software stack and starts the installation process.
- From here on, follow the instructions of the chosen distribution installer. For more information about the installation procedure, refer online installation manuals of the chosen distribution.
- After the installation is complete, a disk image with a random name `<number>.satadisk` will be created in *model-scripts/rdinfra/* folder. Use this disk image for booting the installed distribution.

Additional distribution specific instructions (if any)

Debian

During installation, the installer will prompt the user with the message *'Load CD-ROM drivers from removable media?'* and display two options - *'Yes/No'*. Select the option *'No'*. This is followed by another prompt *'Manually select a CD-ROM module and device?'* and display two options - *'Yes/No'*. Select the option *'Yes'*. This brings up the module list required for accessing CD-ROM and lists two options - *'none'* and *'cdrom'*. Select the option *'none'* and enter `/dev/vda`. The installation media on the virtio disk will be detected and installation continues.

Ubuntu

During installation, the installer will display options for additional packages that may require internet connection, such as *'openssh-server'*. It is important that these options are not selected. Choosing to install additional packages may cause failure to boot after the satadisk image is created.

24.4 UEFI Secure Boot

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Secure boot is a mechanism to build and maintain a complete chain of trust on all the software layers executed in a system and preventing malicious code to be stored and loaded in place of the authenticated one. When the device starts, the firmware checks the signature of each piece of boot software, including UEFI firmware drivers, EFI applications, and the operating system. If the signatures are valid, the device boots, and the firmware gives control to the operating system. Fundamental to the success of the secure boot is the ability to securely store (also referred to as secure storage) and access the keys used for authentication during the various stages of boot.

Secure boot and Secure storage mechanisms are defined by the UEFI specifications. In short, the UEFI specifications define the use of two asymmetric key pairs, platform key (PK) and Key Exchange Key (KEK), and databases for valid and invalid signatures. These keys and databases are used during the secure boot phase which implies that the platform should provide a tamper proof mechanism to store these keys.

The RD platform software allows validation of the secure boot process. This document explains the procedure to build the platform software stack and validate UEFI secure boot on the RD platforms.

Though secure boot process have to be validated using a linux distribution as the target OS, the RD platform software stack currently limits this feature validation to boot of a signed busybox OS.

24.4.1 Generate key pairs

The one-time generation of the following key pairs is mandatory - PK, KEK, DB and DBX. The following commands can be used to generate these key pairs.

- Key Pair Creation : PK, KEK, DB and DBX

```
cd rd-workspace
cd tools/efitools
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=PK/" -keyout PK.key -out PK.crt -
↪days 3650 -nodes -sha256
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=KEK/" -keyout KEK.key -out KEK.
↪crt -days 3650 -nodes -sha256
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=DB_Key/" -keyout DB.key -out DB.
↪crt -days 3650 -nodes -sha256
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=DBX_Key/" -keyout DBX.key -out␣
↪DBX.crt -days 3650 -nodes -sha256
```

- Convert crt certificate to der format

```
openssl x509 -in PK.crt -outform der -out PK.der
openssl x509 -in KEK.crt -outform der -out KEK.der
openssl x509 -in DB.crt -outform der -out DB.der
openssl x509 -in DBX.crt -outform der -out DBX.der
```

The signing of the grub and linux images are performed as a part of build script “build-test-secureboot.sh”. There is no explicit user action required to sign these images.

24.4.2 Build the platform software

The procedure to build the platform software stack for secure boot test is listed below.

To build the software stack, the command to be used is

```
./build-scripts/rdinfra/build-test-secureboot.sh -p <plaform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - Supported commands are
    - clean
    - build
    - package

    * `all` (all of the three above)

Examples of the build command are

- Command to clean, build and package the software stack needed for the secure boot test for RD-N2 platform.

```
./build-scripts/rdinfra/build-test-secureboot.sh -p rdn2 all
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform.

```
./build-scripts/rdinfra/build-test-secureboot.sh -p rdn2 build
```

Note: This command should be followed by the `package` command to complete the preparation of the fip and the disk image.

- Command to package the previously built software stack and prepare the fip and the disk image.

```
./build-scripts/rdinfra/build-test-secureboot.sh -p rdn2 package
```

24.4.3 Securely boot upto Busybox

After the build of the platform software stack for UEFI secure boot is complete, the following command starts the execution of the *selected platform fastmodel* and the software boots up to the busybox prompt. Examples on how to use the command are listed below.

Note: The steps to enroll signatures required to successfully secure boot the platform is listed as well. It is important to execute those steps at least once to validate secure boot support.

To boot up to the busybox prompt, the commands to be used are

- Set `MODEL` path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Launch busybox boot:

```
./secure_boot.sh -p <platform name> -a <additional_params> -n [true|false]
```

Supported command line options are listed below

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -n [true|false] (optional)

  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.

- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to validate the secure boot functionality are as listed below.

- Command to start the execution of the RD-N2 model to boot up to the Busybox prompt with secure boot enabled:

```
./secure_boot.sh -p rdn2
```

- Command to start the execution of the RD-N2 model to boot up to the Busybox prompt with secure boot and network enabled. The model supports virtio.net allowing the software running within the model to access the network:

```
./secure_boot.sh -p rdn2 -n true
```

- Command to start the execution of the RD-N2 model with networking enabled and to boot up to the Busybox prompt with secure boot enabled. Additional parameters to the model are supplied using the -a command line parameter:

```
./secure_boot.sh -p rdn2 -n true -a "-C board.flash0.diagnostics=1"
```

To setup the secure boot process follow the steps listed below on the first boot. Subsequent boots will not need these. Several terminal windows will pop-up in the screen, and the one to interact with has the window title: `FVP terminal_ns_uart_ap`.

1. Interrupt the boot at EDK2 by pressing escape key and dropping into the EDK2 boot menu.
2. Select Device Manager → Secure Boot Configuration → Secure Boot Mode → choose Custom mode and then press enter.
3. Select "Custom Secure Boot Options" and then press enter.
4. Select "DBX Options" → "Enroll Signature" then press enter → "Enroll Signature Using File" and then press enter → Select "NO VOLUME LABEL" and then press enter.
5. Select EFI and press enter → select BOOT and press enter → now Select "DBX.der" and press enter → "Commit Changes and Exit".
6. Repeat steps "4" and "5" for "DB options" for "DB.der".
7. Repeat steps "4" and "5" for "KEK options" for "KEK.der".
8. Repeat steps "4" and "5" for "PK options" for "PK.der".
9. Press Escape and press F10 to save. Ensure that the "Current Secure Boot State" is set as "Enabled".
10. Press Escape and select the "continue" option.
11. Prompts the user to press the "Enter". Press enter key which then reboots the system.

The platform boots up to busybox login prompt with secure boot enabled. If the authentication of the grub or the linux kernel fails, the boot fails and the user is notified about the authentication failure.

To confirm that the boot is indeed a secure boot, the EFI firmware will display messages in the boot log (same window where the secure boot was setup) as shown bellow.

```
Loading driver at 0x000F50A0000 EntryPoint=0x000F676A188
Loading driver at 0x000F50A0000 EntryPoint=0x000F676A188
EFI stub: Booting Linux Kernel...
EFI stub: EFI_RNG_PROTOCOL unavailable, KASLR will be disabled
EFI stub: UEFI Secure Boot is enabled.
EFI stub: Using DTB from configuration table
EFI stub: Exiting boot services and installing virtual address map...
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd490]
```

This completes the validation of the secure boot functionality.

24.5 WinPE Boot

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Neoverse Reference Design (RD) platform software stack supports the boot of Windows Pre-installation Environment (WinPE) on RD platforms. A pre-built WinPE disk image is connected as a SATA disk to the fixed virtual platform (FVP). During boot, the platform firmware detects the connected WinPE disk image and boots from it.

24.5.1 Build the platform software

This section describes the procedure to build the platform firmware required to boot WinPE on Neoverse RD platforms.

To build the RD software stack, the command to be used is

```
./build-scripts/build-test-uefi.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - `clean`
  - `build`
  - `package`
  - `all` (all of the three above)

Examples of the build command are

- Command to clean, build and package the RD-N2 software stack required for WinPE boot on the RD-N2 platform:

```
./build-scripts/build-test-uefi.sh -p rdn2 all
```

- Command to remove the generated outputs (binaries) of the software stack for the RD-N2 platform:

```
./build-scripts/build-test-uefi.sh -p rdn2 clean
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform:

```
./build-scripts/build-test-uefi.sh -p rdn2 build
```

Note: This command should be followed by the `package` command to complete the preparation of the fip image.

- Command to package the previously built software stack and prepares the fip image:

```
./build-scripts/build-test-uefi.sh -p rdn2 package
```

24.5.2 Obtain the WinPE disk image

Obtain a pre-built WinPE disk image to use it as the disk image to boot from. Refer to this page for more information.

Note: WinPE version should be `20262` or higher.

24.5.3 Boot WinPE

To boot from the WinPE disk image, the commands to be used are:

- Set `MODEL` path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Launch the FVP to boot WinPE:

```
./distro.sh -p <platform name> -d <satadisk_path> -a <additional_params> -n␣
↪[true|false]
```

Supported command line options are listed below

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -d <satadisk_path>
  - Absolute path to the WinPE disk image created using the previous section.
- -n [true|false] (optional)

  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.

- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to boot WinPE are as listed below.

- Command to begin the WinPE boot on the RD-N2 platform using a `WinPE_arm64.iso` pre-built disk image. Follow the instructions on console to complete the WinPE boot.

```
./distro.sh -p rdn2 -d /absolute/path/to/WinPE_arm64.iso
```

CHAPTER

TWENTYFIVE

COMPUTE EXPRESS LINK

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Compute Express Link (CXL) is an open standard interconnection for high-speed central processing unit (CPU)-to-device and CPU-to-memory, designed to accelerate next-generation data center performance. CXL is built on the PCI Express (PCIe) physical and electrical interface with protocols in three key areas: input/output (I/O), memory, and cache coherence.

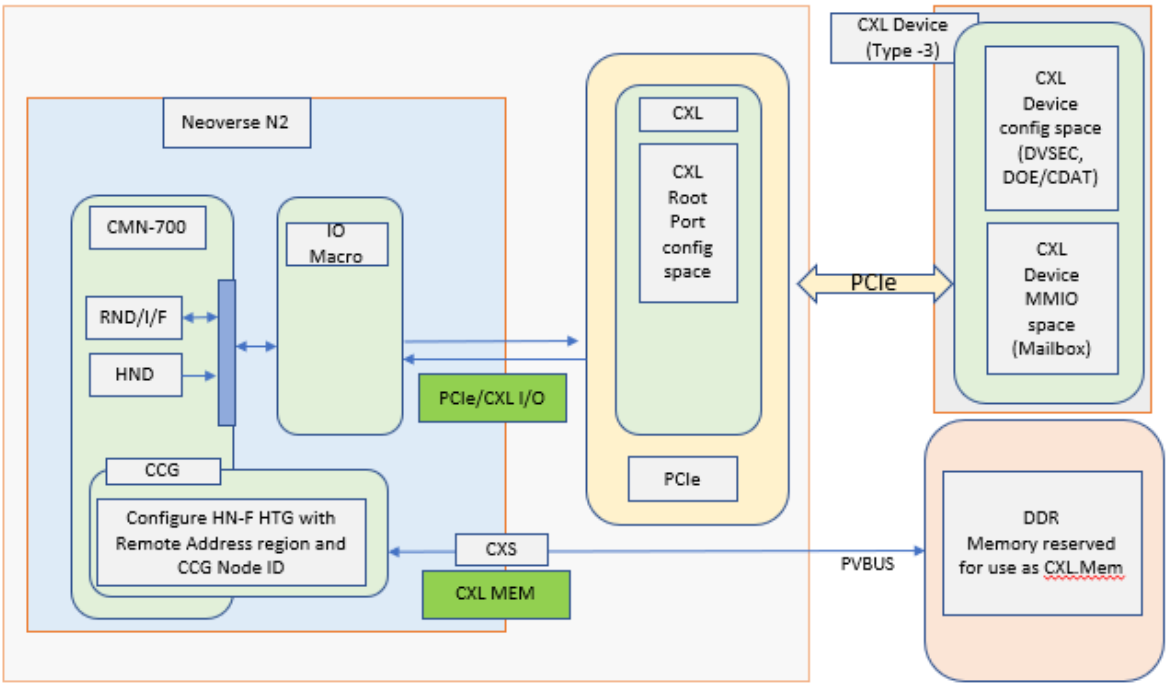


Fig. 25.1: CXL Type-3 device modeled on Neoverse N2 reference design platform.

This document explains CXL 2.0 Type-3 device (Memory expander) handling on Neoverse N2 reference design platform. At present, CXL support has been verified on 'rdn2cfg1' platform. CXL Type-3 device supports CXL.io and CXL.mem protocol and acts as a Memory expander to the Host SOC.

25.1 CXL Software Overview

System Control Processor (SCP) firmware

1. At Host address space 8GB address space, starting at, 3FE_0000_0000h is reserved for CXL Memory. This address space is part of SCG and configured as Normal cacheable memory region.
2. CMN-700 is the main interconnect, which will be configured for PCIe enumeration and topology discovery.
3. pcie_enumeration module performs PCIe enumeration and as part of the enumeration process it is also checked whether a PCIe device supports CXL Extended Capability. pcie_enumeration module invokes CXL module API to determine the same for each of the detected PCIe device.
4. CXL module will also determine whether CXL device has DOE capability. Once found, execute DOE operations to fetch CDAT structure and understand CXL device memory range supported. DOE operation sequence is implemented following DOE-ECN 12Mar-2020.

   Check for CXL object's DOE busy bit and initiate DOE operation accordingly for fetching CXL CDAT Structures(DSMAS supported at latest FVP model). Read the CXL device DPA base, DPA length from DSMAS structures and save the same into internal Remote Memory software Data Structure.
5. After completing the enumeration process pcie_enumeration module would invoke CXL module API to map remote CXL memory region into Host address space and do necessary CMN configuration.

   Software data structure for remote memory will have information regarding CXL Type-3 Device Physical memory address, size and memory attributes. CXL module would call CMN module API for doing the necessary interconnect configuration.
6. CMN module configures HN-F Hashed Target Region(HTG) with the address region reserved for Remote CXL Memory usage, based on the discovered remote device memory size. Configured HN-F CCG SA node IDs and CXL.Mem region in HNF-SAM HTG in following order-

   ```
   HNF_SAM_CCG_SA_NODEID_REG
   HNF_SAM_HTG_CFG3_MEMREGION
   HNF_SAM_HTG_CFG2_MEMREGION
   HNF_SAM_HTG_CFG1_MEMREGION
   ```

   Program por_ccg_ra_sam_addr_region_reg. with target HAID, host memory base address and size for accessing remote CXL memory.

EDK2 Platform

1. A new CXL.Dxe is introduced that looks for PCIe device with CXL and DOE capability. This discovery process begins based on notification received on installation of gEfiPciEnumerationCompleteProtocolGuid.
2. It first looks for PCIe devices with extended capability and then check whether the device supports DOE. If DOE operation is supported then send DOE command and get remote memory details in the form of CDAT tables (DSMAS). The operation is similar to what is done in SCP firmware, that's explained above.
3. After enumerating complete PCIe topology, all remote memory node details will be stored in local data structure and CXLPlatformProtocol interface will be installed.
4. ACPITableGenerator module dynamically prepares ACPI tables. It will use CXLPlatformProtocol interfaces and get the previously discovered remote CXL memory details. It would prepare SRAT table with both Local memory, remote CXL memory nodes, along with other necessary details.

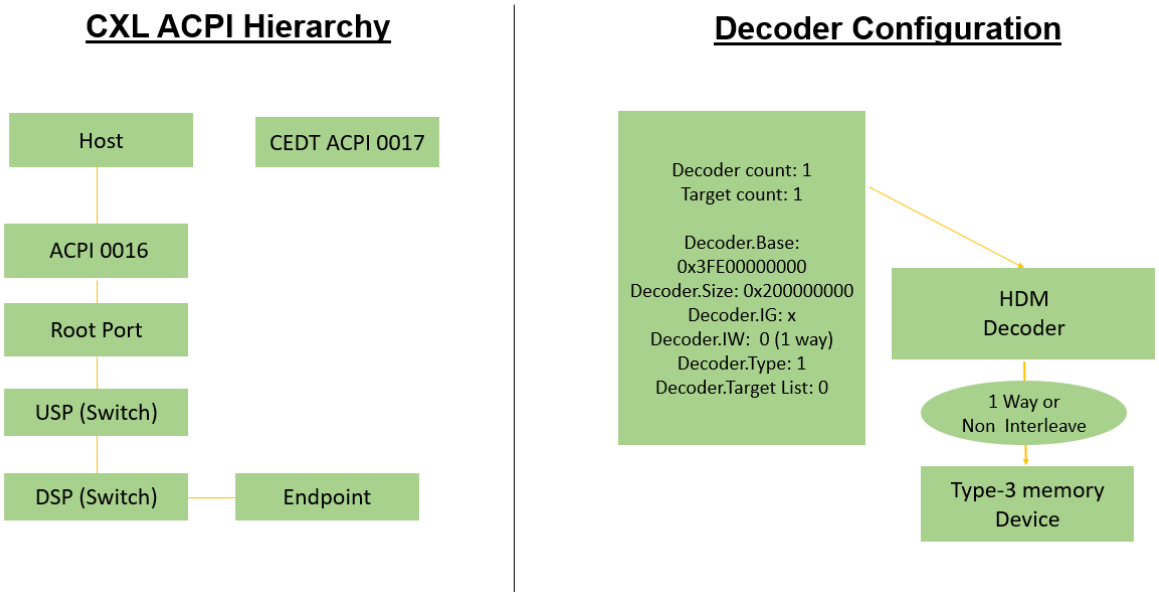
   Prepare HMAT table with required proximity, latency info.
5. The remote CXL memory will be represented to kernel as Memory only NUMA node.

6. Also, CEDT structures, CHBS and CFMWS are created and passed to kernel. In CFMWS structure, Interleave target number is considered 1 for demonstrating a reference solution with CEDT structures in the absence of interleaving capability in current FVP model. There is no real interleaving address windows across multiple ports with this configuration. It is same as single port CXL Host bridge.
7. ACPI0016 and ACPI0017 objects are created using PcieAcpiTableGenerator.Dxe at runtime and passed to kernel. ACPI0016 would indicated the presence of CXL Host bridge and ACPI0017 would correspond to CMFWS and CHBS structures.

Kernel

1. All firmware work is validated using CXL framework present in Kernel.

25.2 CXL with CEDT and Decoder configuration



25.3 Download and build the required platform software

For downloading and building the platform firmware, refer *Buildroot boot* or *Busybox Boot*. Any other boot mechanism, like Distro boot may also be fine for CXL capability test.

Ensure that the model parameter "-C pcie_group_0.pciex16.pcie_rc.add_cxl_type3_device_to_default_hierarchy=true" is present in "rdinfra/platforms/<rd platform>/run_model.sh"

25.4 Validating CXL capabilities in Kernel

In following explanation, 'buildroot' boot is taken as an example. With buildroot there are more utility options available.

1. Boot the platform to buildroot command line prompt.
2. Run the command 'lspci -k', which will list out the all PCIe devices and associated kernel driver. Showing below, the output for CXL device. Please note that BDF position of CXL device may vary based on the PCIE topology of the model.

```
00:18.0 Memory controller [0502]: ARM Device ff82 (rev 0f)
Subsystem: ARM Device 000f
Kernel driver in use: cxl_pci
```

   One point to note here that ensure CXL is enabled in kernel 'defconfig'.

```
CONFIG_CXL_BUS=y
CONFIG_CXL_MEM_RAW_COMMANDS=y
```

3. As a next command to check the capabilities of CXL device, execute 'lspci -vv -s 00:18.0', which would display following output.

```
00:18.0 Memory controller [0502]: ARM Device ff82 (rev 0f) (prog-if 10)
  Subsystem: ARM Device 000f
  Control: I/O- Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr- Stepping-␣
↪SERR- FastB2B- DisINTx-
  Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >
↪SERR- <PERR- INTx-
  IOMMU group: 10
  Region 0: Memory at 60800000 (32-bit, non-prefetchable) [size=64K]
  Capabilities: [40] Power Management version 1
          Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot+,D3cold-)
          Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=0 PME-

  ....

  Capabilities: [118 v1] Extended Capability ID 0x2e
  Capabilities: [130 v1] Designated Vendor-Specific: Vendor=1e98 ID=0000 Rev=1␣
↪Len=40: CXL
          CXLCap: Cache- IO+ Mem+ Mem HW Init- HDMCount 1 Viral-
          CXLCtl: Cache- IO+ Mem- Cache SF Cov 0 Cache SF Gran 0 Cache Clean- Viral-
          CXLSta: Viral-
  Capabilities: [158 v1] Designated Vendor-Specific: Vendor=1e98 ID=0008 Rev=0␣
↪Len=20 <?>
  Kernel driver in use: cxl_pci
```

4. For checking the CXL device memory capabilities NUMA utilities can be used. Enable NUMACTL package in buildroot 'defconfig'.

```
For example, in 'configs/rdn2cfg1/buildroot/aarch64_rdinfra_defconfig' enable 'BR2_
↪PACKAGE_NUMACTL=y'
```

With NUMA utilities available in buildroot, execute command 'numactl -H', which would show all the available NUMA nodes and it's capacities.

```
numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 7930 MB
node 0 free: 7824 MB
node 1 cpus:
node 1 size: 8031 MB
node 1 free: 8010 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

Here it shows that Node-1(CXL device) has memory capacity of 8031MB, which adds to the total available memory for the system. This extended memory regions is available for kernel usage, which can be verified using NUMA utilities 'numademo', 'numastat'.

```
#numastat -n

Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit                 215.21           84.72          299.93
Numa_Miss                  0.00            0.00            0.00
Numa_Foreign               0.00            0.00            0.00
Interleave_Hit            25.98           26.68           52.66
Local_Node               215.21            0.00          215.21
Other_Node                 0.00           84.72           84.72
```

5. If NUMA utilities are not present then CXL device memory information can be verified using numa node1 sysfs entries.

```
[ceoss@localhost ~]$ cat /sys/devices/system/node/node1/meminfo
Node 1 MemTotal:        8224032 kB
Node 1 MemFree:         8203836 kB
Node 1 MemUsed:           20196 kB
Node 1 Active:                0 kB
Node 1 Inactive:              0 kB
...
Node 1 KReclaimable:       2180 kB
Node 1 Slab:               6060 kB
Node 1 SReclaimable:       2180 kB
Node 1 SUnreclaim:         3880 kB
Node 1 HugePages_Total:     0
Node 1 HugePages_Free:      0
```

(continues on next page)

(continued from previous page)

```
Node 1 HugePages_Surp:      0
```

Above examples demonstrate how CXL Type-3 device is used as Memory expander and the device memory region can be utilized by kernel.

25.5 CEDT and CXL ACPI configuration in Kernel sysfs

1. Checking CXL mem device size through CXL sysfs interface. (Showing the CXL.Mem device size 8GB)

```
# cat /sys/bus/cxl/devices/mem0/ram/size
  0x200000000
```

2. CXL Mem device at root device downstream port.

```
# cat /sys/bus/cxl/devices/root0/dport0/physical_node/0000\:00\:18.0/mem0/ram/
↪size
  0x200000000
```

3. Decoder configurations passed through CFMWS and seen in kernel.

```
# cat /sys/bus/cxl/devices/root0/decoder0.0/start
  0x3fe00000000

# cat /sys/bus/cxl/devices/root0/decoder0.0/size
  0x200000000

# cat /sys/bus/cxl/devices/root0/decoder0.0/target_list
  0

# cat /sys/bus/cxl/devices/root0/decoder0.0/interleave_ways
  1
```

CHAPTER

TWENTYSIX

MCP SIDEBAND CHANNEL

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

26.1 Overview

Server systems generally hosts a myriad of IPs/controllers within a single system. For Neoverse Reference Design platforms, this could range from SCP (system control processor), AP (application processor), MCP (manageability control processor) etc. In production environments, an additional board management controller would also be added to the system which would help system administrators to monitor the live status of the system remotely. In such environments, communication between these controllers and especially between the BMC and other components is of utmost importance. The system administrator can rely on the status of the system from the BMC only if such a reliable communication enables these controllers to talk to the BMC. MCP sideband channel feature aims at show-casing one such means of communication using `PLDM<->MCTP` protocol stack.

SBMR specification recommends certain guidelines in using these stacks to implement the MCP sideband channel. Care has been taken to align to the specification in areas where it was feasible to do so. However, please note that Neoverse Reference Design platforms doesn't support a BMC and therefore all the communication as of now is implemented via a loopback on MCP itself. MCP has been chosen as the controller for showcasing the feature as one of its core responsibilities is to to communicate and share information with the BMC. If the system supports sensors and effecters with little or no-intelligence to it, the MCP can read and write data from them and transfer them over to the BMC.

26.2 What does MCP sideband channel showcase?

MCP sideband channel show-cases packet transactions over a `PLDM<->MCTP` stack based system implemented on MCP. Packets are sent and received on MCP over a loopback interface which mimics the physical layer.

Firmware on MCP has been segregated to implement a MCP terminal and a BMC terminal. The feature show-cases BMC as the primary terminal trying to discover information about the secondary terminal, MCP. `PLDM discovery`, as quoted in the PLDM specification has been implemented in firmware. BMC terminal uses `PLDM discovery` to send out request packets to figure out the terminal ID, PLDM types, PLDM commands and the version for these commands. MCP also holds a dummy PDR record. A PDR record could be thought of as a block of semantic information required to understand how sensor/effecter data on a particular terminal could be parsed at a node remote to the one that forms it. In our example, if we assume the MCP terminal to be connected to a sensor, it is essential for the BMC terminal to understand how to read/parse the sensor data. MCP terminal is required to form PDR records and transfer it to the BMC terminal on request to aid in this scenario. The dummy PDR held by the MCP terminal is retrieved as part of `PLDM discovery`.

For more information on the design of firmware, refer to *MCP sideband channel design*

26.3 Building and running MCP sideband channel

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Follow the instructions as given in *Busybox Boot* to boot linux busybox on the platform.

The feature is setup in such a way that the BMC firmware automatically starts off with `PLDM discovery` to identify and retrieve information from MCP terminal. On MCP controller's uart terminal, you should be able to see the logs starting with the following prints.

```
[    0.000000] [MCP]: mcp context:
[    0.000000] [PLDM_FW]:
[    0.000000] [PLDM_FW]: pldm tid:                        0
[    0.000000] [PLDM_FW]: pldm type count:                 2
[    0.000000] [PLDM_FW]: global enable:                   0
[    0.000000] [PLDM_FW]: receiver addr:                   0
[    0.000000] [PLDM_FW]: heartbeat timer:                 0
[    0.000000] [PLDM_FW]: transport protocol type:         0
```

26.4 Decoding output logs

The output logs can be decoded as follows.

The first part of the logs point to the PLDM terminal information for MCP terminal within the segregated firmware. This represents the PLDM TID and different sets of PLDM features supported on the MCP terminal. At this point, the BMC terminal is yet to discover this information.

```
[    0.000000] [MCP]: mcp context:
[    0.000000] [PLDM_FW]:
[    0.000000] [PLDM_FW]: pldm tid:                        0
[    0.000000] [PLDM_FW]: pldm type count:                 2
[    0.000000] [PLDM_FW]: global enable:                   0
[    0.000000] [PLDM_FW]: receiver addr:                   0
[    0.000000] [PLDM_FW]: heartbeat timer:                 0
[    0.000000] [PLDM_FW]: transport protocol type:         0
[    0.000000] [PLDM_FW]: pldm type:                       0
[    0.000000] [PLDM_FW]: version count:                   1
[    0.000000] [PLDM_FW]: version[0] :             f1.f0.f0.0
[    0.000000] [PLDM_FW]: commands count:                  4
[    0.000000] [PLDM_FW]: command[0]:                      2
[    0.000000] [PLDM_FW]: command[1]:                      3
[    0.000000] [PLDM_FW]: command[2]:                      4
[    0.000000] [PLDM_FW]: command[3]:                      5
[    0.000000] [PLDM_FW]: pldm type:                       2
[    0.000000] [PLDM_FW]: version count:                   1
[    0.000000] [PLDM_FW]: version[0] :             f1.f2.f0.0
```

(continues on next page)

(continued from previous page)

```
[    0.000000] [PLDM_FW]: commands count:                3
[    0.000000] [PLDM_FW]: command[0]:                   49
[    0.000000] [PLDM_FW]: command[1]:                   57
[    0.000182] [PLDM_FW]: command[2]:                   81
```

This is followed by the BMC terminal initiating the actual `PLDM discovery`.

```
[    0.000282] [BMC]: pldm discovery start ...
```

What follows is a set of `PLDM<->MCTP` based requests and responses to transfer MCP terminal's PLDM terminal information. Each command transaction involves 2 cycles of `PLDM<->MCTP` stack walk. This is better explained in the *MCP sideband channel design* section. For brevity, a small snippet of the transaction has been pasted below.

- `A request being sent`

```
[    0.000482] [MCTP]: sending pkt, len 8
[    0.000607] [LOOPBACK]: sending packet onto loopback bus
[    0.000982] [LOOPBACK]: receiving packet from loopback bus
```

- `Corresponding response being sent`

```
[    0.001082] [MCTP]: sending pkt, len 9
[    0.001214] [LOOPBACK]: sending packet onto loopback bus
[    0.001388] [LOOPBACK]: receiving packet from loopback bus
```

- `Similar cycle for other PLDM commands`

```
[    0.001482] [MCTP]: sending pkt, len 12
[    0.001682] [LOOPBACK]: sending packet onto loopback bus
[    0.001822] [LOOPBACK]: receiving packet from loopback bus
[    0.001982] [MCTP]: sending pkt, len 7
[    0.002082] [LOOPBACK]: sending packet onto loopback bus
[    0.002182] [LOOPBACK]: receiving packet from loopback bus

[    0.002382] [MCTP]: sending pkt, len 17
[    0.002482] [LOOPBACK]: sending packet onto loopback bus
[    0.002582] [LOOPBACK]: receiving packet from loopback bus
[    0.002782] [MCTP]: sending pkt, len 44
[    0.002882] [LOOPBACK]: sending packet onto loopback bus
[    0.003037] [LOOPBACK]: receiving packet from loopback bus
```

Once all transactions are done, the discovery completes gracefully.

```
[    0.007282] [PLDM_FW]: pldm discovery complete
```

Finally, BMC terminal prints all the data it received from MCP terminal. This has to match with the prints put out by MCP terminal before the transactions started.

```
[    0.007482] [PLDM_FW]:
[    0.007549] [PLDM_FW]: pldm tid:                     0
[    0.007682] [PLDM_FW]: pldm type count:              2
[    0.007782] [PLDM_FW]: global enable:                2
[    0.007982] [PLDM_FW]: receiver addr:                8
[    0.008082] [PLDM_FW]: heartbeat timer:              0
```

(continues on next page)

(continued from previous page)

```
[    0.008282] [PLDM_FW]: transport protocol type:        0
[    0.008382] [PLDM_FW]: pldm type:                      0
[    0.008503] [PLDM_FW]: version count:                  1
[    0.008682] [PLDM_FW]: version[0] :           f1.f0.f0.0
[    0.008850] [PLDM_FW]: commands count:                 4
[    0.008982] [PLDM_FW]: command[0]:                     2
[    0.009082] [PLDM_FW]: command[1]:                     3
[    0.009284] [PLDM_FW]: command[2]:                     4
[    0.009382] [PLDM_FW]: command[3]:                     5
[    0.009482] [PLDM_FW]: pldm type:                      2
[    0.009718] [PLDM_FW]: version count:                  1
[    0.009805] [PLDM_FW]: version[0] :           f1.f2.f0.0
[    0.009982] [PLDM_FW]: commands count:                 3
[    0.010152] [PLDM_FW]: command[0]:                    49
[    0.010282] [PLDM_FW]: command[1]:                    57
[    0.010412] [PLDM_FW]: command[2]:                    81
```

The fields `global enable`, `receiver addr`, `heartbeat timer` and `transport protocol type` could hold different values on BMC terminal when compared to MCP terminal. `receiver addr` corresponds to the address of BMC terminal and rest of fields corresponds to configurations that enable events that BMC terminal is interested in. This data is send to the MCP terminal from the BMC terminal along the discovery process to let the MCP terminal know what all events it is interested in receiving notification from and the address to which those events needs to be forwarded. At the time when MCP context is printed, these fields are not yet set.

In addition to the PLDM information that BMC terminal has received, a PDR record has also been received (rather retrieved) by the BMC terminal. This is the last set of data to appear in the logs. The PDR record is printed as raw bytes here.

```
[    0.010582] [PLDM_FW]: pdr [0]:
[    0.010682] [PLDM_FW]:                                 1
[    0.010782] [PLDM_FW]:                                 0
[    0.010882] [PLDM_FW]:                                 0
[    0.011082] [PLDM_FW]:                                 0
[    0.011193] [PLDM_FW]:                                 2
[    0.011382] [PLDM_FW]:                                 3
[    0.011540] [PLDM_FW]:                                 4
[    0.011682] [PLDM_FW]:                                 0
[    0.011800] [PLDM_FW]:                                 5
[    0.011982] [PLDM_FW]:                                 0
[    0.012082] [PLDM_FW]:                                 6
[    0.012182] [PLDM_FW]:                                 0
[    0.012408] [PLDM_FW]:                                 7
[    0.012494] [PLDM_FW]:                                 0
[    0.012682] [PLDM_FW]:                                 8
[    0.012842] [PLDM_FW]:                                 0
[    0.012982] [PLDM_FW]:                                 9
[    0.013082] [PLDM_FW]:                                 0
[    0.013282] [PLDM_FW]:                                10
[    0.013382] [PLDM_FW]:                                 0
[    0.013482] [PLDM_FW]:                                11
[    0.013709] [PLDM_FW]:                                12
[    0.013782] [PLDM_FW]:                                13
[    0.013982] [PLDM_FW]:                                14
```

26.5 MCP sideband channel design

PLDM<->MCTP transactions are in a way analogous to TCP/IP transaction for any application protocol. Take the example of an FTP server running over TCP/IP. FTP, the application layer deals with transferring chunks of file data as packets. Further, we have TCP as the transport layer underneath which deals with fragmentation, re-ordering, acknowledgment of receipt etc to make sure the transport went through well. Similarly PLDM acts as the application layer. PLDM specification dictates what data to transferred in each packet. MCTP is the transport layer. Like TCP, it deals with fragmentation and re-ordering.

Following PLDM commands have been used in the in the feature.

PLDM Command	PLDM Type	Code Value
GetTID	PLDM BASE	0x02
GetPLDMVersion	PLDM BASE	0x03
GetPLDMTypes	PLDM BASE	0x04
GetPLDMCommands	PLDM BASE	0x05
SetEventReceiver	PLDM PLATFORM	0x04
GetPDR	PLDM PLATFORM	0x51

PLDM specification defines the request and response formats for each of these commands. To better understand the transactions, GetTID could be taken as an example. BMC terminal forms the GetTID PLDM packet and transfers it to MCTP layer. MCTP forwards the command to the loopback interface which sends the packet to itself. Loopback receiver then forwards the packet to MCTP which forwards it to the MCP terminal. This could be thought as the first cycle or the request cycle.

MCP terminal decodes the packet, forms the response and sends it back to MCTP. The packet essentially traverses one more cycle until it finally reaches BMC terminal. This could be thought of as the second cycle or the response cycle. For multi-part transactions, the number of cycles to complete one command transfer may not be limited to two cycles.

MCP sideband channel software makes use of the following specifications.

- PLDM Base specification
- PLDM Platform specification
- PLDM Codes
- PLDM over MCTP Binding
- MCTP specification

Following thrid party libraries also have been used.

- libpldm
- libmctp

CHAPTER

TWENTYSEVEN

MEMORY SYSTEM RESOURCE PARTITIONING AND MONITORING (MPAM)

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

27.1 MPAM-resctrl - A quick glance

MPAM stands for memory system resource partitioning and monitoring. As the name suggests, it deals with two things; partitioning and monitoring. MPAM's resource partitioning logic deals with partitioning resources such as shared CPU caches, interconnect caches, memory bandwidth, interconnect bandwidth, etc. In MPAM terminology, such resources can be classified as MSCs. How each MSC gets partitioned varies from MSC to MSC and also by the type of MSC. For instance, partitioning a cache could be very different from partitioning memory bandwidth. MPAM's resource monitoring logic deals with monitoring each MSC. A monitor can measure resource usage or capacity usage, depending on the resource. For instance, a cache can have monitors for cache storage that measures the usage of the cache. Reading a monitor could help in tuning the memory-system partitioning controls. For detailed information on MPAM, refer to MPAM specification

resctrl is a Linux kernel feature by which Arm's MPAM and Intel's RDT can be configured and controlled. resctrl exposes MPAM capabilities and configuration options via a file-system interface. On the latest kernel source tree, users would find resctrl adapted for X86 RDT. The file and folder names reflect RDT's feature sets rather than a generic resource portioning interface naming or MPAM's feature names. In short, for Arm64 architecture, resctrl is how the user space can configure MPAM. The steps by which MPAM could be configured via resctrl are described in the subsequent section.

27.2 Exploring resctrl file-system

MPAM-resctrl is enabled by default on the platform (from here on platform/ platform under test/ platform under consideration would be abbreviated as `PuT`). This documentation advises users to follow the *Busybox* build to enable MPAM-resctrl capabilities for the PuT. Once the necessary sources have been fetched, checkout `RD-INFRA-2024.07.15-MPAM` tag for linux repository. Additionally, build-option "LINUX_TOOLS_IOMMU_BUILD" has to be set to "0" from build-scripts/configs/<platform>/<platform> file before proceeding with the build. Build and boot the system to command prompt. Run the following command to mount the resctrl file-system. It is to be noted that MPAM's performance aspect cannot be tested on an FVP, rather only the register configurations could be tested on it.

```
# mount -t resctrl resctrl /sys/fs/resctrl
```

It would be good to refer to resctrl documentation in parallel as many of the concepts that would be discussed further along would be present in better clarity in the documentation. However, as mentioned in the beginning, be aware that the documentation as of now covers resctrl file-system as utilized by Intel's RDT.

Once resctrl file-system has been mounted, change directory to /sys/fs/resctrl and list the files.

```
# cd /sys/fs/resctrl
/sys/fs/resctrl# ls

cpus        info     mon_data      schemata    tasks
cpus_list   mmode    mon_groups    size
```

These are the files and folders through which MPAM's MSCs for the PuT would be accessed and configured. Before proceeding further, it is important to understand more about MPAM's PARTID. PARTID can be considered as an ID or label associated with MPAM configurations for a single software environment or a collection of software environments. Quoting MPAM specification "An MPAM resource control uses the PARTID that is set for one or more software environments. A PARTID for the current software environment labels each memory system request. Each MPAM resource control has control settings for each PARTID. The PARTID in a request selects the control settings for that PARTID, which are then used to control the partitioning of the performance resources of that memory-system component". In short, each set of MPAM configuration is associated with a PARTID. The required configuration is selected/modified by programming the associated PARTID into MPAMCFG_PART_SEL register present at the MSC's memory-mapped interface.

MPAM driver is designed in such a way that the default configuration uses a single PARTID (PARTID 0) with the default maximum partition configuration for the MSCs. This is done in the early stages of Linux kernel boot up. This will be covered in greater detail in the sections to come.

resctrl is organized in such a way that each PARTID would in turn have a separate copy of all these files and folders. At this point, there is just one set of these files/folders as shown above. More the number of PARTIDs, more would be the copy of these sets of files and folders. To understand what these files/folders denote, the user could try the following.

```
/sys/fs/resctrl # cat cpus
ffff

/sys/fs/resctrl # cat cpus_list
0-15
```

The file named `cpus` lists CPUs having access to the MPAM's MSCs under consideration, for a given PARTID. The output is in bitmap format. For the PuT, it shows 0xffff indicating the presence of 16 CPUs. Reading contents of the file named `cpus_list` shows the same information in a different style (CPUs marked from 0-15).

```
/sys/fs/resctrl # cat schemata
L3:49=ffff
```

`schemata` would be one of the most important files out of the list of files exposed by resctrl. It shows the MPAM resource, its ID and the partition for this particular PARTID. From the above logs, it is clear that the MSC to be partitioned is an L3 cache, having cache ID 49. The default cache portion bitmask assigned for this PARTID is '0xffff' which means the entire cache.

As discussed earlier in the *MPAM-resctrl - A quick glance* section, an MSC is partitioned in accordance with its type. When it comes to caches, two partitioning schemes can be used - cache portion partitioning and cache capacity partitioning. For cache portion partitioning, a cache is divided into equal number of portions represented by a bitmap. A '1' indicates that the corresponding portion is allowed and '0' otherwise. 0xffff represents the cache portion bitmap with all portions enabled. Since cache capacity partitioning is not being exercised here, this won't be discussed in this documentation. Please refer to MPAM specification to get a better idea about these partitioning schemes.

Neoverse reference design platforms as of now don't have an L3 cache. Instead, system level cache (SLC) on the

interconnect acts as the shared cache for all DSU clusters. SLC cache for the PuT has been added within the PPTT table. The cache topology parsing logic within the OS walks through all caches available associates each cache with a level. SLC caches for the PuT is mapped as an L3 cache. For more details, refer to PPTT and MPAM ACPI tables present in the source code.

```
/sys/fs/resctrl # cat tasks
1
2
3
4
~
```

Reading the `tasks` file would give an idea of the tasks that use this PARTID. Writing a task id to the file will add a task to the group. Since this is the default config, the user should be able to find all the tasks in this file. An example where the `tasks` file gets modified will be looked at in the latter part of this section.

```
/sys/fs/resctrl # cat mode
shareable
```

The `mode` of the resource group dictates the sharing of its allocations. A "shareable" resource group allows sharing of its allocations while an "exclusive" resource group does not allow sharing.

```
/sys/fs/resctrl # cd info
/sys/fs/resctrl/info # ls
L3    L3_MON    last_cmd_status
```

The `info` directory contains information about the enabled resources. Each resource has its own sub-directory. There should be a sub-directory with the name that reflects the resource's names. Since SLC has been modeled as an L3 MPAM node, an `L3` directory should be present. If the resource supports monitoring capabilities, a folder with the name `<MSC>_MON` should also exist. `L3_MON` in this case is the directory having information about L3's monitoring capabilities.

```
/sys/fs/resctrl/info # cd L3
/sys/fs/resctrl/info/L3 # ls

bit_usage     min_cbm_bits     shareable_bits
cbm_mask      num_closids
```

`L3` sub-directory contains the files as shown above. Enter the following commands to understand what each of these files denote.

```
/sys/fs/resctrl/info/L3 # cat cbm_mask
ffff
```

`cbm_bitmask` shows the cache portion bitmask corresponding to 100% allocation of the MSC. This value is in line with what is observed as the cache portion bitmap given in `schemata`.

```
/sys/fs/resctrl/info/L3 # cat bit_usage
49=XXXXXXXXXXXXXXXX
```

`bit_usage` gives details about how each instance of the MSC gets used. Since `schemata` describes the cache portion bitmap for L3, `bit_usage` talks about the status of each of these portions. Each portion represented by a bit could be any of the below types.

`0`: Corresponding region is unused. When the system's resources have been allocated and a "0" is found in "bit_usage" it is a sign that resources are wasted.

H: Corresponding region is used by hardware only but available for software use. If a resource has bits set in "shareable_bits" but not all of these bits appear in the resource groups' schematas then the bits appearing in "shareable_bits" but no resource group will be marked as "H".

X: Corresponding region is available for sharing and used by hardware and software. These are the bits that appear in "shareable_bits" as well as a resource group's allocation.

S: Corresponding region is used by software and available for sharing.

E: Corresponding region is used exclusively by one resource group. No sharing allowed.

P: Corresponding region is pseudo-locked. No sharing is allowed.

From the value that is read out, all 16 portions of the cache portion bitmap are of type shareable.

```
/sys/fs/resctrl/info/L3 # cat min_cbm_bits
1
```

min_cbm_bits denotes the minimum number of consecutive bits which must be set when writing a mask. Setting anything lower than what min_cbm_bits suggests would lead to an error.

```
/sys/fs/resctrl/info/L3 # cat shareable_bits
ffff
```

shareable_bits is again a bitmask of all the shareable bits in the cache portion bitmask. For the PuT, it is 0xffff.

```
/sys/fs/resctrl/info/L3 # cat num_closids
32
```

num_closid denotes the number of closids. closids again is Intel's terminology which expands to "class of service IDs". This essentially means PARTIDs under MPAM. Therefore, num_closid tells us the number of valid PARTIDs the MSC supports.

```
/sys/fs/resctrl/info # cat last_cmd_status
ok
```

At the top level of the info directory, there is a file named last_cmd_status. This is reset with every "command" issued via the file-system (making new directories or writing to any of the control files). If the command was successful, it will read as "ok". If the command fails, it will provide more information about the error generated during the operation. A simple example is shown below.

```
/sys/fs/resctrl # echo L3:49=0000 > schemata
sh: write error: Invalid argument

/sys/fs/resctrl # cat info/last_cmd_status
Mask out of range
```

As discussed earlier, the min_cbm_mask or the minimum bitmask that should be programmed into the configuration register is at least 1. If a value less than min_cbm_mask is used, the resctrl filesystem would throw an error.

27.3 Configuring MPAM via resctrl file-system

The file-system interface for the default PARTID has been looked at in the last section. Real MPAM use-cases have multiple partition spaces (PARTIDs) with different MSC partitions. With resctrl, adding a new partition space (PARTID) is simple; create a new folder with any name (users are advised to give a name resonating the use-case so that maintenance becomes easier) in the root resctrl directory.

```
/sys/fs/resctrl # mkdir partid_space_2
/sys/fs/resctrl # ls

cpus                    mode                partid_space_2      tasks
cpus_list               mon_data            schemata
info                    mon_groups          size
```

```
/sys/fs/resctrl # cd partid_space_2/
/sys/fs/resctrl/partid_space_2 # ls

cpus                    mode                mon_groups          size
cpus_list               mon_data            schemata            tasks
```

Once a new folder named `partid_space_2` is created, MPAM driver internally allocates a new PARTID and associates it with this new resctrl directory. The user can modify the configurations via the resctrl file-system. resctrl talks with the MPAM driver and the driver would in turn program the required configuration registers for the new PARTID for the MSC under consideration to add the new configurations. In order to define the `schemata` for this new PARTID, do the following.

```
/sys/fs/resctrl/partid_space_2 # cat schemata
L3:49=ffff

/sys/fs/resctrl/partid_space_2 # echo "L3:49=3ff" > schemata
/sys/fs/resctrl/partid_space_2 # cat schemata
L3:49=03ff
```

As shown above, to define a `schemata`, a file write to the schemata file under the new PARTID's root directory is required. Whenever a new folder is added under the resctrl root directory, the `schemata` would always reflect the default maximum for the resource under consideration - in this case, the L3 cache with 0xffff. The value to be written has to align with the format by which `schemata` describes the MSC and its partitions. In this case, the new value should be of the format `L3:<cache ID>=<cache portion bitmap>`. Changing the schemata of the default PARTID space is also valid. Users could try changing the value of the default schemata as an experiment.

As the new `schemata` values have been updated, the next step would be to update the `tasks` file with the tasks that need to use this new partitioning scheme. Select one task at random from `ps -A`.

```
/sys/fs/resctrl/partid_space_2 # cat tasks
/sys/fs/resctrl/partid_space_2 #
/sys/fs/resctrl/partid_space_2 # ps -A

PID   USER     TIME  COMMAND
1 0         0:00 sh
2 0         0:00 [kthreadd]
3 0         0:00 [rcu_gp]
~
23 0         0:00 [kworker/2:0H-ev]
```

(continues on next page)

(continued from previous page)

```
24 0          0:00 [cpuhp/3]
25 0          0:00 [migration/3]
```

For this demonstration, task 23 has been selected to be added to the new PARTID/ partition space. Before assigning the task, take a look at the `tasks` file under the default PARTID to make sure that the task is currently assigned to it. As discussed in the beginning, with just the default PARTID, all tasks should be part of the default PARTID's `task` file.

```
/sys/fs/resctrl/partid_space_2 # cd ../
/sys/fs/resctrl # cat tasks

1
2
3
4
~
23
24
~
```

Proceed to add the task to the `tasks` file under `partid_space_2`.

```
/sys/fs/resctrl # cd partid_space_2
/sys/fs/resctrl/partid_space_2 # echo 23 > tasks
/sys/fs/resctrl/partid_space_2 # cat tasks

23
```

A task can any time exist only under one configuration. This means that the task would no longer be present under the default PARTID's `tasks` directory.

```
/sys/fs/resctrl/partid_space_2 # cd ../
/sys/fs/resctrl # cat tasks

1
2
3
4
~
24
~
```

Additional tasks can be added to the `tasks` file in the same manner by which the first task was added.

```
/sys/fs/resctrl # cd partid_space_2
/sys/fs/resctrl/partid_space_2 # echo 24 > tasks
/sys/fs/resctrl/partid_space_2 # cat tasks

23
24
```

Multiple PARTIDs up to `num_closid` limit can be added in the same fashion. Repeat the steps to configure the schemata and tasks as shown above for any new PARTID directory created.

```
/sys/fs/resctrl # cd ../
/sys/fs/resctrl # mkdir partid_space_3
/sys/fs/resctrl # ls

cpus                    mode                partid_space_2      tasks
cpus_list               mon_data            schemata            size
partid_space_3          info                mon_groups
```

```
/sys/fs/resctrl # cd partid_space_3/
/sys/fs/resctrl/partid_space_3 # ls

cpus                    mode                mon_groups          size
cpus_list               mon_data            schemata            tasks
```

27.4 A closer look at MPAM software

Enabling MPAM on the PuT involves enabling MPAM EL1/EL2 register access from EL3 (trusted firmware), building kernel drivers and having proper ACPI tables to populate platform-specific MPAM data.

```
EFI_ACPI_6_4_PPTT_STRUCTURE_CACHE_INIT (
  PPTT_CACHE_STRUCTURE_FLAGS,           /* Flag */
  0,                                    /* Next level of cache */
  SIZE_8MB,                             /* Size */
  8192,                                 /* Num of sets */
  16,                                   /* Associativity */
  PPTT_UNIFIED_CACHE_ATTR,              /* Attributes */
  64,                                   /* Line size */
  RD_PPTT_CACHE_ID(0, -1, -1, L3Cache)  /* Cache id */
)
```

For processor side caches, MPAM references the cache/MSC of interest via cache ID. The way an MSC gets referenced in the MPAM table changes from MSC to MSC. Please refer to MPAM ACPI Specification to get a detailed understanding of how MPAM tables are described. For a complete view of the PPTT table implemented on the PuT, please refer to Platform/ARM/SgiPkg/AcpiTables/<PuT>/Pptt.aslc under uefi/edk2/edk2-platforms repository in the source files. Corresponding MPAM ACPI table entries, based on MPAM ACPI v2.0 are as shown below.

```
/* MPAM_MSC_NODE 1 */
{
  RD_MPAM_MSC_NODE_INIT(0x1, RDN2CFG1_BASE_ADDRESS(0x141601000, 0),
     RDN2CFG1_MPAM_MMIO_SIZE, 0, RDN2CFG1_MPAM_MSC_COUNT,
     RDN2CFG1_RESOURCES_PER_MSC,
     RDN2CFG1_FUNCTIONAL_DEPENDENCY_PER_RESOURCE)
},

/* MPAM_MSC_NODE 2 */
{
  RD_MPAM_MSC_NODE_INIT(0x2, RDN2CFG1_BASE_ADDRESS(0x141641000, 0),
     RDN2CFG1_MPAM_MMIO_SIZE, 0, RDN2CFG1_MPAM_MSC_COUNT,
     RDN2CFG1_RESOURCES_PER_MSC,
     RDN2CFG1_FUNCTIONAL_DEPENDENCY_PER_RESOURCE)
},
```

It is to be noted that the above snippet has references to one of the platform that supports MPAM. As the PuT changes, the references and variable names would also change. The number of SLC cache slices can vary on each platform. Each of the cache slice would be configured as an MSC. Unique indices should be used for each SLC slice as OS would use the index as one of the criteria to differentiate between MSC nodes. For a complete view of MPAM ACPI table, please refer to Platform/ARM/SgiPkg/AcpiTables/<PuT>/Mpam.aslc file under uefi/edk2/edk-plaforms repository in the source files.

On the Linux side, MPAM software can be categorized into MPAM ACPI driver, MPAM platform driver, MPAM platform devices, MPAM layer for resctrl, MPAM support for architecture, etc. This would not be the complete list, but still covers most of the major software layers MPAM touches.

Quite early into the Linux boot, `__init_el2_mpam` (arch/arm64/include/asm/ el2_setup.h) is invoked from within `head.S`. `__init_el2_mpam` takes care of detecting and MPAM, doing basic MPAM system register setup and trap disablement to EL2.

```
.macro __init_el2_mpam
#ifdef CONFIG_ARM64_MPAM
    /* Memory Partioning And Monitoring: disable EL4 traps */
    mrs     x1, id_aa64pfr0_el1
    ubfx    x0, x1, #ID_AA64PFR0_MPAM_SHIFT, #4
    cbz     x0, 1f                          // skip if no MPAM
    msr_s   SYS_MPAM0_EL1, xzr              // use the default partition..
    msr_s   SYS_MPAM2_EL2, xzr              // ..and disable lower traps
    msr_s   SYS_MPAM1_EL1, xzr
    mrs_s   x0, SYS_MPAMIDR_EL1
    tbz     x0, #17, 1f                     // skip if no MPAMHCR reg
    msr_s   SYS_MPAMHCR_EL2, xzr            // clear TRAP_MPAMIDR_EL1 -> EL2
    1:
#endif /* CONFIG_ARM64_MPAM */
.endm
```

As the kernel proceeds to boot, the MPAM platform driver initialization routine gets invoked (`mpam_msc_driver_init`). The total count of MPAM MSCs is queried from the MPAM ACPI table. This is also the first time the MPAM ACPI table gets queried, starting from kernel boot up. Platform driver would get initialized only if a valid MPAM ACPI table with at least one MSC is defined. Once the platform driver is initialized, MPAM driver probing kicks off (`mpam_msc_drv_probe`). It is at this point that the MPAM ACPI table is completely parsed and appropriate platform device data structures are populated. Each of the populated MSC gets registered as an individual platform device. Once all the platform devices are probed, temporary CPU hotplug callbacks (`mpam_discovery_cpu_online`) are installed. if the system supports 128 MSCs, the callbacks would only get registered after the 128th platform device gets registered. The callbacks installed at this point are for discovering hardware details about MSCs (known as hardware probing in MPAM driver terminology) and would be replaced at a later point. This is the reason why they are described as temporary callbacks. More information on CPU hotplugging and supported API sets can be found at CPU hot plugging on Linux. Please refer to drivers/platform/mpam/mpam_devices.c under the Linux kernel repository to see the detailed implementation of the routines discussed here.

Soon after the CPU hotplug callbacks are installed, the corresponding `setup` (`mpam_discovery_cpu_online`) callbacks get called by each of the CPUs. Suppose if the PuT has 16 CPUs, the `setup` function would be called 16 times with CPU IDs ranging from 0-15. At this stage, `setup` callback proceeds with MSC hardware discovery. This includes discovering details such as the features supported, maximum PARTID, maximum PMG, etc. To understand all the features a particular MSC could support, please refer to MPAM Specification chapter 9. Once the supported features are discovered and maximum PARTID and PMG values supported are established, a default config is programmed to the configuration registers (MPAMCFG_*) for each of these features for all PARTIDs starting from 0 to the maximum value. `setup` function is defined in such a way that the first CPU to come online would discover features of all the registered MSCs and program appropriate configs for them. Rest of the `setup` calls on the other CPUs would skip over hardware discovery. A small snippet of what happens in the `setup` function (`mpam_discovery_cpu_online`) is shown below.

```
/* For all MSCs, if the current CPU has access to the MSC and HW discovery
 * is yet to be carried out for the MSC under consideration, proceed with
 * the discovery.
 */

list_for_each_entry(msc, &mpam_all_msc, glbl_list) {
    if (!cpumask_test_cpu(cpu, &msc->accessibility))
        continue;

    spin_lock(&msc->lock);
    if (!msc->probed)
      err = mpam_msc_hw_probe(msc);
    spin_unlock(&msc->lock);

    if (!err)
      new_device_probed = true;
```

The logic to program any config register (MPAMCFG_*) has been mentioned in MPAM specification, section 11.1.2.

After the first CPU to come up completes hardware probing and feature configuration, the kernel is free to enable MPAM. This is done with the help of workqueues.

```
static DECLARE_WORK(mpam_enable_work, &mpam_enable);

~

if (new_device_probed && !err)
    schedule_work(&mpam_enable_work);
```

The code scheduled under the workqueue shown above gets executed soon after the probing. This is where MPAM resctrl configurations are set up. resctrl has a dependency with cacheinfo and hence the workqueue task that's responsible for setting up resctrl stays in wait state until cacheinfo is up and ready. cacheinfo deals with populating cache nodes from PPTT and exporting them to /sys/devices/system/cpu/cpu*/cache/index* for user space to access. MPAM's resctrl layer internally queries the MSC cache node's size from cacheinfo and thus have to wait till proper data is available.

```
wait_event(wait_cacheinfo_ready, cacheinfo_ready);

~

static int __init __cacheinfo_ready(void)
{
    cacheinfo_ready = true;
    wake_up(&wait_cacheinfo_ready);

    return 0;
}
device_initcall_sync(__cacheinfo_ready);
```

A `teardown` (`mpam_cpu_offline`) callback is also part of the hotplug callbacks installed earlier. The `teardown` callback gets called when the CPUs go offline. Atomic reference counters are added within the data structures that manage each MSC. In case of a hotplug shutdown on the PuT, the MPAM driver wouldn't reprogram any register or initiate cleanup until the last CPU goes offline.

```
list_for_each_entry_rcu(msc, &mpam_all_msc, glbl_list) {
if (!cpumask_test_cpu(cpu, &msc->accessibility))
    continue;

spin_lock(&msc->lock);
if (msc->reenable_error_ppi)
    disable_percpu_irq(msc->reenable_error_ppi);

if (atomic_dec_and_test(&msc->online_refs))
    mpam_reset_msc(msc, false);
spin_unlock(&msc->lock);
```

Once cacheinfo is set up, MPAM's resctrl setup proceeds. With the completion of resctrl, MPAM is ready to be enabled and a new set of hotplug callbacks are installed replacing the old one. The maximum PARTID and PMG that the system can support have been established at this point and can't be changed after the new callbacks are installed.

```
/*
 * Once the cpuhp callbacks have been changed, mpam_partid_max can no
 * longer change.
 */
 spin_lock(&partid_max_lock);
 partid_max_published = true;
 spin_unlock(&partid_max_lock);

 static_branch_enable(&mpam_enabled);
 mpam_register_cpuhp_callbacks(mpam_cpu_online);
```

As discussed earlier, the new `setup` function deals with marking CPUs online and reprogramming MSCs in case all CPUs went down. Just like the `teardown` function, the first CPU to come up would re-program the feature registers for each PARTID. The same atomic reference counter used in the `teardown` function is used here for this purpose.

```
rcu_read_lock();
list_for_each_entry_rcu(msc, &mpam_all_msc, glbl_list) {
if (!cpumask_test_cpu(cpu, &msc->accessibility))
    continue;

spin_lock(&msc->lock);
if (msc->reenable_error_ppi)
    _enable_percpu_irq(&msc->reenable_error_ppi);

if (atomic_fetch_inc(&msc->online_refs) == 0)
    mpam_reprogram_msc(msc);
spin_unlock(&msc->lock);
}
rcu_read_unlock();

if (mpam_is_enabled())
    mpam_resctrl_online_cpu(cpu);
```

Once the system boots up and resctrl is mounted, PARTID 0 with default maximum cache portion bitmap comes into use. Whenever a new directory is added, the MPAM driver selects the new PARTID to be the first free PARTID in a range of PARTIDs from 0 to maximum. More information about the PARTID allocator could be found from fs/resctrl/rdtgroup.c within the kernel source tree. Since the file-system interface is tied to Intel's feature set and convention, PARTID allocator is named as `closid_allocator`.

```
for_each_set_bit(closid, &closid_free_map, closid_free_map_len) {
   if (IS_ENABLED(CONFIG_RESCTRL_RMID_DEPENDS_ON_CLOSID) &&
       resctrl_closid_is_dirty(closid))
           continue;

   clear_bit(closid, &closid_free_map);
   return closid;
}
```

Also, for every folder created, a default config needs to be programmed into MPAM's supported MSC's feature configuration registers for the new PARTID. For PuT, this means programming L3's cache portion bitmaps with the default maximum portion bitmap. This is also taken care of by resctrl. The snippet below shows a portion of the MPAM driver API code that gets called when a new folder is created.

```
case RDT_RESOURCE_L3:
cfg.cpbm = cfg_val;
mpam_set_feature(mpam_feat_cpor_part, &cfg);
break;

~
return mpam_apply_config(dom->comp, partid, &cfg);
```

The same API (`mpam_apply_config`) is used when the user makes any change in the `schemata`. Instead of the default config, the cache portion bitmap written by the user gets programmed into the MPAM configuration register for the PARTID.

Even if the L3 cache/SLC for the PuT supports a large set of PARTIDs, resctrl has a limit of 32 PARTIDs at maximum due to the bitmaps algorithm used for closid calculation. If the user tries to generate more than 32 folders including the root folder /sys/fs/resctrl, the system would throw an error.

```
/*
 * MSC may raise an error interrupt if it sees an out or range partid/pmg,
 * and go on to truncate the value. Regardless of what the hardware
 * supports, only the system wide safe value is safe to use.
 */
u32 resctrl_arch_get_num_closid(struct rdt_resource *ignored)
{
    return min((u32)mpam_partid_max + 1, (u32)RESCTRL_MAX_CLOSID);
}
```

Please refer to drivers/platform/mpam/mpam_resctrl.c in the Linux source tree to get a detailed understanding of MPAM's interaction with resctrl.

27.5 MPAM and task scheduling

In the last section, the main focus was on understanding how the MPAM driver was designed, how the resctrl file-system interacted with the MPAM driver and the basic boot initialization sequence of the MPAM driver. In this section, an interesting topic would be looked at; how MPAM works along with the task scheduler.

Once MPAM is enabled, each task should belong to a PARTID group. Since PARTID gets so tightly ingrained with a task's basic identity, the `thread_info` (arch/arm64/include/asm/thread_info.h) struct has been modified to hold an additional member as shown below.

```
/*
 * low level task data that entry.S needs immediate access to.
 */
 struct thread_info {
 ~

 #ifdef CONFIG_ARM64_MPAM
 u64    mpam_partid_pmg;
 #endif
```

When a system boots up with MPAM enabled and resctrl mounted, all tasks belong to the default PARTID-PMG (0) group. Once new partitions are allocated and tasks are moved from one PARTID-PMG group to another, this member of the `thread_info` (`mpam_partid_pmg`) would have to be updated accordingly. Below is the stack dump for the case where a task is moved from the default PARTID group to a new one.

```
/sys/fs/resctrl/partid_space_2 # echo 26 > tasks

[  404.607377] CPU: 3 PID: 1 Comm: sh Not tainted 5.17.0-g5bf032719b99-dirty
↪#19
[  404.607381] Hardware name: ARM LTD RdN2Cfg1, BIOS EDK II Jun 15 2022
[  404.607384] Call trace:
[  404.607386]  dump_backtrace.part.0+0xd0/0xe0
[  404.607391]  show_stack+0x1c/0x6c
[  404.607396]  dump_stack_lvl+0x68/0x84
[  404.607400]  dump_stack+0x1c/0x38
[  404.607405]  resctrl_arch_set_closid_rmid+0x50/0xac
[  404.607410]  rdtgroup_tasks_write+0x2b0/0x4a0
[  404.607414]  rdtgroup_file_write+0x24/0x40
[  404.607419]  kernfs_fop_write_iter+0x11c/0x1ac
[  404.607424]  new_sync_write+0xe8/0x184
[  404.607427]  vfs_write+0x230/0x290
[  404.607431]  ksys_write+0x68/0xf4
[  404.607435]  __arm64_sys_write+0x20/0x2c
[  404.607439]  invoke_syscall+0x48/0x114
[  404.607444]  el0_svc_common.constprop.0+0x44/0xec
[  404.607449]  do_el0_svc+0x28/0x90
[  404.607453]  el0_svc+0x20/0x60
[  404.607457]  el0t_64_sync_handler+0x1a8/0x1b0
[  404.607461]  el0t_64_sync+0x1a0/0x1a4
```

The write to `tasks` file ends up as a synchronous exception from a 64-bit lower EL. The exception handler then routes it to the appropriate resctrl routines which then proceeds to call `resctrl_arch_set_closid_rmid`. On taking a closer look at `resctrl_arch_set_closid_rmid`, it takes care of calling `mpam_set_cpu_defaults` with the new PARTID and PMG. `mpam_set_cpu_defaults` goes ahead to update the `thread_info` member field of the very task that got swapped between PARTID groups.

```
void resctrl_arch_set_cpu_default_closid_rmid(int cpu, u32 closid, u32 pmg)
{
    BUG_ON(closid > U16_MAX);
    BUG_ON(pmg > U8_MAX);

    if (!cdp_enabled) {
        mpam_set_cpu_defaults(cpu, closid, closid, pmg, pmg);
```

(continues on next page)

(continued from previous page)

```
~
```

```
static inline void mpam_set_task_partid_pmg(struct task_struct *tsk,
                                            u16 partid_d, u16 partid_i,
                                            u8 pmg_d, u8 pmg_i)
{
#ifdef CONFIG_ARM64_MPAM
    u64 regval;

    regval = FIELD_PREP(MPAM_SYSREG_PARTID_D, partid_d);
    regval |= FIELD_PREP(MPAM_SYSREG_PARTID_I, partid_i);
    regval |= FIELD_PREP(MPAM_SYSREG_PMG_D, pmg_d);
    regval |= FIELD_PREP(MPAM_SYSREG_PMG_I, pmg_i);

    WRITE_ONCE(task_thread_info(tsk)->mpam_partid_pmg, regval);
#endif
}
```

How would the `mpam_partid_pmg` field from `thread_info` get utilized? The actual use of this field is in enabling the propagation of corresponding PARTID-PMG value pair via the bus interface downstream. Every memory request should be tagged with PARTID-PMG fields so that the MSCs downstream can respond according to the feature configuration that has been set up on it for that particular PARTID-PMG that it received from upstream. For PuT, PARTID-PMG would be propagated downstream via the CHI interface. To enable propagation of PARTID-PMG values, the system register `MPAM0_EL1` have to be programmed with the PARTID-PMG value. From MPAM specification, this register's purpose is described as follows - "Holds information to generate MPAM labels for memory requests when executing at EL0." Please refer to the MPAM Specification chapter 4 to get detailed information on MPAM information propagation.

When the system boots up with all tasks in the default configuration, the PARTID-PMG pair would have a value of zero and `MPAM0_EL1` would hold this same value. The early boot call to `__init_el2_mpam` writes zero to this system register. As new PARTIDs are allocated and tasks are moved from the default PARTID group, `MPAM0_EL1` would need re-programming. When a task that had been moved from the default group to a new group gets scheduled, there has to be a check to see if the PARTID-PMG pair that `MPAM0_EL1` holds is the one that `thread_info` for the task that got scheduled has. `mpam_thread_switch` (arch/arm64/include/asm/mpam.h) does the exact same thing.

```
__notrace_funcgraph __sched
struct task_struct *__switch_to(struct task_struct *prev,
                                struct task_struct *next)
{
struct task_struct *last;
~

/*
 * MPAM thread switch happens after the DSB to ensure prev's accesses
 * use prev's MPAM settings.
 */
mpam_thread_switch(next);
```

```
static inline void mpam_thread_switch(struct task_struct *tsk)
{
    u64 oldregval;
    int cpu = smp_processor_id();
```

(continues on next page)

(continued from previous page)

```
    u64 regval = mpam_get_regval(tsk);

    if (!IS_ENABLED(CONFIG_ARM64_MPAM))
        return;

    if (!static_branch_likely(&mpam_enabled))
        return;


    oldregval = READ_ONCE(per_cpu(arm64_mpam_current, cpu));
    if (oldregval == regval)
        return;

    if (!regval)
        regval = READ_ONCE(per_cpu(arm64_mpam_default, cpu));

    write_sysreg_s(regval, SYS_MPAM0_EL1);
    WRITE_ONCE(per_cpu(arm64_mpam_current, cpu), regval);
}
```

Every time a task switch happens via `__switch_to`, `mpam_thread_switch` gets called with the new `task_struct` (include/linux/sched.h) struct as param . What has been programmed in `MPAM0_EL1` for the CPU in context, is held in an `SMP` specific per CPU variable called `arm64_mpam_current`. If there is a mismatch between the `thread_info` value and the value in `MPAM0_EL1`, the value from `thread_info` is copied to `MPAM0_EL1`. Re-programming the value in `MPAM0_EL1` generally happens when two tasks of different PARTID-PMG group gets scheduled on the same core. If the tasks keep switching back and forth on the CPU in context, the system register keeps getting programmed with relevant PARTID-PMG pairs.

To conclude, a simple test done on the PuT would be discussed below. As part of the test, a new PARTID (partid_space_2) space was created as soon as the system booted to prompt. A simple script that moved tasks from the default PARTID space to the new PARTID space was used to move tasks under `partid_space_2`.

```
/sys/fs/resctrl/partid_space_2 # cat  ~/mv_task.sh

#/bin/sh!

for i in `seq $1 $2`

do
    echo "$i" > tasks
done
```

Basic conditional debug logs were added in the build within `mpam_thread_switch`. The process list was dumped to get an idea of the processes that were planned to be moved to the new PARTID space (PID 5 to 20).

```
/sys/fs/resctrl/partid_space_2 # ps -A

PID   USER     TIME  COMMAND
 1 0         0:00 sh
 2 0         0:00 [kthreadd]
 3 0         0:00 [rcu_gp]
 4 0         0:00 [rcu_par_gp]
 6 0         0:00 [kworker/0:0H]
```

(continues on next page)

(continued from previous page)

```
 8 0         0:00 [mm_percpu_wq]
 9 0         0:00 [rcu_tasks_kthre]
10 0         0:00 [ksoftirqd/0]
11 0         0:00 [rcu_preempt]
12 0         0:00 [migration/0]
13 0         0:00 [cpuhp/0]
14 0         0:00 [cpuhp/1]
15 0         0:00 [migration/1]
16 0         0:00 [ksoftirqd/1]
17 0         0:00 [kworker/1:0-mm_]
18 0         0:00 [kworker/1:0H]
19 0         0:00 [cpuhp/2]
20 0         0:00 [migration/2]
```

The following logs were observed as soon as the tasks were moved from the default PARTID space to the new PARTID space.

```
/sys/fs/resctrl/partid_space_2 #  ~/mv_task.sh 5 20

[  274.393977] oldregval (arm64_mpam_current) : 0         //chunk 1
[  274.393977] regval (thread_info field)     : 10001
[  274.393981] pid                            : 11
[  274.393981] tgid                           : 11
[  274.393981] cpu id                         : 1
[  274.393984] SYS_MPAM0_EL1 before update    : 0
[  274.393987] SYS_MPAM0_EL1 after update     : 10001
[  274.393987]
[  274.393991] oldregval (arm64_mpam_current) : 10001     //chunk 2
[  274.393993] regval (thread_info field)     : 0
[  274.393996] pid                            : 0
[  274.393999] tgid                           : 0
[  274.393999] cpu id                         : 1
[  274.401977] SYS_MPAM0_EL1 before update    : 10001
[  274.401980] SYS_MPAM0_EL1 after update     : 0
[  274.401983]
[  274.401985] oldregval (arm64_mpam_current) : 0         //chunk 3
[  274.401985] regval (thread_info field)     : 10001
[  274.401990] pid                            : 11
[  274.401992] tgid                           : 11
[  274.401995] cpu id                         : 1
[  274.401998] SYS_MPAM0_EL1 before update    : 0
[  274.401998] SYS_MPAM0_EL1 after update     : 10001
[  274.409975]
[  274.409978] oldregval (arm64_mpam_current) : 10001     //chunk 4
[  274.409980] regval (thread_info field)     : 0
[  274.409983] pid                            : 0
[  274.409983] tgid                           : 0
[  274.409987] cpu id                         : 1
[  274.409990] SYS_MPAM0_EL1 before update    : 10001
[  274.409992] SYS_MPAM0_EL1 after update     : 0
[  274.409995]
```

The above log can be divided into 4 chunks of data, each captured at the time when one of the threads were being

scheduled. The first chunk shows the `tgid`, a value equivalent to the PID which is visible from user space, being scheduled on CPU 1. Since we moved PID 11 to the new partid space, `partid_space_2` with PARTD 1, the new PARTID-PMG value stored in its `thread_info` field, `mpam_partid_pmg` would be `10001`. However, the last thread scheduled on this CPU was of PARTID 0 group as indicated by the per-CPU variable (`oldreg`) in the logs. This is the same value stored in MPAM0_EL1. Since there is a mismatch between these values, MPAM0_EL1 is updated with the new PARTID-PMG pair using the `WRITE_ONCE` macro to avoid store tearing and re-ordering.

The next chunk shows that the thread with `tgid`/PID 0 gets scheduled on the same CPU. However, PID 0 still belongs to the default PARTID space and thus there is a conflict between its `thread_info` field and the newly programmed PARTID-PMG value in `MPAM0_EL1`/`arm64_mpam_current`. The default PARTID-PMG again gets programmed into `MPAM0_EL1` and `arm64_mpam_current`. Two more context switches have been captured, where chunk 3 is similar to chunk 1 and chunk 4 to chunk 2. Kernel changes for MPAM are quite large and for brevity, what is most essential only has been covered in this documentation.

CHAPTER

TWENTYEIGHT

POWER MANAGEMENT

28.1 ACPI Low Power Idle (LPI)

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

28.1.1 Overview of LPI test

ACPI Low Power Idle (LPI) mechanism allows an operating system to manage the power states of the processor power domain hierarchy. Neoverse Reference Design platforms support the c-states c0 (run state), c1 (WFI) and c3 (WFI with core powered down).

This document describes the procedure to validate LPI functionality, determining the number of times a particular CPU core switched to idle state and the total time the core has been in a idle state.

28.1.2 Download and build the required platform software

For downloading and building the platform firmware, refer *Buildroot boot*. To enable LPI from ACPI, update the `LPI_EN` variable from `SgiPlatform.dsc.inc` before build. Also remember to enable stress-ng binary from the buildroot config.

28.1.3 Procedure for validating LPI states

1. Boot the platform to buildroot command line prompt.
2. Run the command 'nproc' to get the cpu count in the system.
3. Read the idle state descriptor entry to know about the c-state information.

```
cat /sys/devices/system/cpu/cpu<x>/cpuidle/state<j>/desc

Here, x = 0, 1, 2, ... (nproc -1)
      y = 0, 1, 2, ...
```

generally for RD platform:

state0: c1 (LPI1) state for CPUx

state1: c3 (LPI3) state for CPUx

state2: available only for plaforms having power control for CPU container and is the combined c3 (LPI3 for core and LPI2 for cluster) state for CPU and cluster.

4. To get the LPI statistics, read the 'usage' and 'time' entries:

```
cat /sys/devices/system/cpu/cpu<x>/cpuidle/state<y>/usage
cat /sys/devices/system/cpu/cpu<x>/cpuidle/state<y>/time
```

5. Wake up all CPUs from sleep. The example shown below uses the 'stress-ng' utility. Run stress-ng utility for one second for all CPUs using the command

```
stress-ng -c <num_cpu> -t 1

Here num_cpu is the value obtained on step 2
```

6. Repeat step 4 and compare the usage and time values.

In a system with idle states enabled, the expectation is the 'usage' count should increment on each suspend-resume cycle. The value for 'time' specifies the total time period the core was in that particular state.

Note: In a system that supports state2, the usage count will increment for either state1 or for state2. This is applicable when a core is the last one to undergo sleep inside a container, then the core will request for a combined sleep state instead of core only power down.

28.2 Collaborative Processor Performance Control (CPPC)

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

28.2.1 Overview of CPPC test

Collaborative Processor Performance Control (CPPC) is a mechanism for the OS to manage the performance of the processor core on a contiguous and abstract performance scale. The CPPC support as implemented for Neoverse Reference Design platforms requires the CPUs to support the Arm v8.4 AMU functionality. So the support for CPPC is applicable for platforms that have Arm v8.4 or higher CPU.

The CPPC kernel framework has two parts, monitoring the CPU performance and scale the CPU performance. In the monitoring part, the OS uses the AMU extension which is introduced in ARMv8.4. Especially the 'Processor frequency counter' and the 'Constant frequency counter'. For calculating the processor frequency, the values of the processor frequency counter and constant frequency counter are captured at two instances, say 2 microseconds between the instances and get the delta between these two counts. The constant frequency is known hence the processor frequency is calculated as:

```
(delta processor frequency count / delta constant frequency count)
        * constant frequency
```

In the controlling part, the OS requests the desired performance to the platform firmware through a non-secure channel between the OS and platform firmware.

This document focus on the procedure to validate CPPC functionality, obtaining the CPU's current operating frequency, procedure to scale CPU frequency and the scaling governor.

28.2.2 Download and build the required platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

For downloading and building the platform firmware, refer *Busybox Boot* or *Buildroot boot* documentation. To enable CPPC from ACPI, update the `CPPC_EN=TRUE` variable from `SgiPlatform.dsc.inc` before build.

Note: For *Busybox Boot* or *Buildroot boot*, in Linux kernel defconfig ensure `CONFIG_ACPI_CPPC_CPUFREQ=y`.

28.2.3 Changing the scaling governor

For changing the frequency governor, the procedure is:

1. Boot the platform to command line prompt.
2. Read the scaling governor entry to get the current governor in action.

```
cat /sys/devices/system/cpu/cpufreq/policy<x>/scaling_governor

For RD platforms, x = 0, 1, 2, ... (number of CPUs - 1)
```

3. Read the scaling available governors entry to get list of supported governors.

```
cat /sys/devices/system/cpu/cpufreq/policy_x/scaling_available_governors
```

4. To change governor, write the preferred governor to scaling governor entry.

```
echo governor_name > /sys/devices/system/cpu/cpufreq/policy<x>/scaling_governor

Here the governor_name is obtained from step 3.
```

5. Repeat step 3 to confirm the governor change is taken into effect.

28.2.4 Validating CPPC functionality

For validating the CPPC functionality, it is recommended to use 'userspace' governor. The procedure for validation is:

1. Set 'userspace' governor as the scaling governor.

```
echo userspace > /sys/devices/system/cpu/cpufreq/policy<x>/scaling_governor
```

2. Write the desired frequency in KHz to the scaling setspeed entry.

```
echo freq_in_KHz > /sys/devices/system/cpu/cpufreq/policy<x>/scaling_set_speed

For RD-V1 variants, the supported frequencies in GHz are 1.3, 1.5, 1.7, 2.1 and 2.6
For RD-N2 variants, the supported frequencies in GHz are 2.3, 2.6 and 3.2
For RD-V3 variants, the supported frequencies in GHz are 1.7, 2.0, 2.3, 2.6, 2.9␣
↪and 3.2
```

3. Read the cpuinfo current frequency entry, to obtain the current operating frequency of the CPU, using the AMU extension.

28.2.5 Additional precautions for FVP based platforms

The CPPC frequency monitoring part should be executed with highest time precision. For FVP based platforms, to improve the time precision, follow the steps below.

1. Export these variables before launching the model

```
export FM_SCX_ENABLE_TIMER_LOCAL_TIME=1
export FASTSIM_DISABLE_TA=0
export FASTSIM_AUTO_SYNC=1
```

2. Pass `--quantum=1` as model parameter.
3. For single-chip platforms, pass `--min-sync-latency=0` and for multichip platforms, pass `--min-sync-latency=1` also as model parameter.

28.3 Reboot and Shutdown

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

28.3.1 Overview of the reboot modes supported

RD-V3 platform supports:

1. Shutdown: Upon receiving a shutdown request, the application processor (AP) conveys it to the System Control Processor (SCP) through the SCMI communication channel, initiating a dynamic power-down for the AP cores. Subsequently, the SCP executes the shutdown sequence, notifying the Runtime Security Engine (RSE) and then the Manageability Control Processor (MCP). In response to the shutdown request, the RSE undertakes power-down activities and enters the Wait for Interrupt (WFI) mode. Meanwhile, the MCP performs its power-down procedures and communicates with the designated entity responsible for system shutdown (e.g., BMC controller), which is responsible for configuring the Power Management Integrated Circuit (PMIC). On RD-V3 FVP platforms, when the MCP UART print the shutdown message, the FVP platform terminates its operations.
2. Cold reboot: Upon receiving a cold reboot request, the AP signals it to SCP through the SCMI communication channel, and then initiate the dynamic power-down for the AP cores. Subsequently, the SCP executes the power-down sequence, notify the MCP and then RSE. In response to the cold reboot request, MCP undertakes power-down activities and enter WFI mode. Meanwhile the RSE will perform its power-down activities and issue a system wide reset by programming the system reset register. This will reset the entire system including RSE, SCP, MCP, LCP and AP cores.
3. Warm reboot: In the warm reboot mode, designed for an Application Processor (AP) exclusive reboot, the AP signals the System Control Processor (SCP) through the SCMI communication channel, initiating a dynamic power-down sequence for the AP cores. Upon receiving the warm reboot request, the SCP places the AP cores in a static power-off mode by programming the Power Processing Units (PPUs). After ensuring that all cores are statically powered down, the SCP proceeds to power up the boot CPU.

Impact of power-down on various components:

	RSE	SCP	MCP	LCP	AP
Shutdown	OFF	OFF	OFF	OFF	OFF
Cold reboot	Reset	Reset	Reset	Reset	Reset
Warm reboot	NILL	NILL	NILL	NILL	Reset

28.3.2 Power-down sequence for RD-V3 platform

AP side

The power-down sequence remains consistent for shutdown, cold reboot, and warm reboot at the Application Processor (AP) end.

Upon receiving the power-down request, the Linux kernel performs cleanup activities and utilizes the Symmetric Multiprocessing (SMP) framework to transition all secondary CPUs into Wait for Interrupt (WFI) mode. The primary CPU leverages the EFI reboot runtime service to initiate a PSCI call with the power-down mode, resulting in a context switch on the AP side. CPU0 transitions from non-secure world to root world.

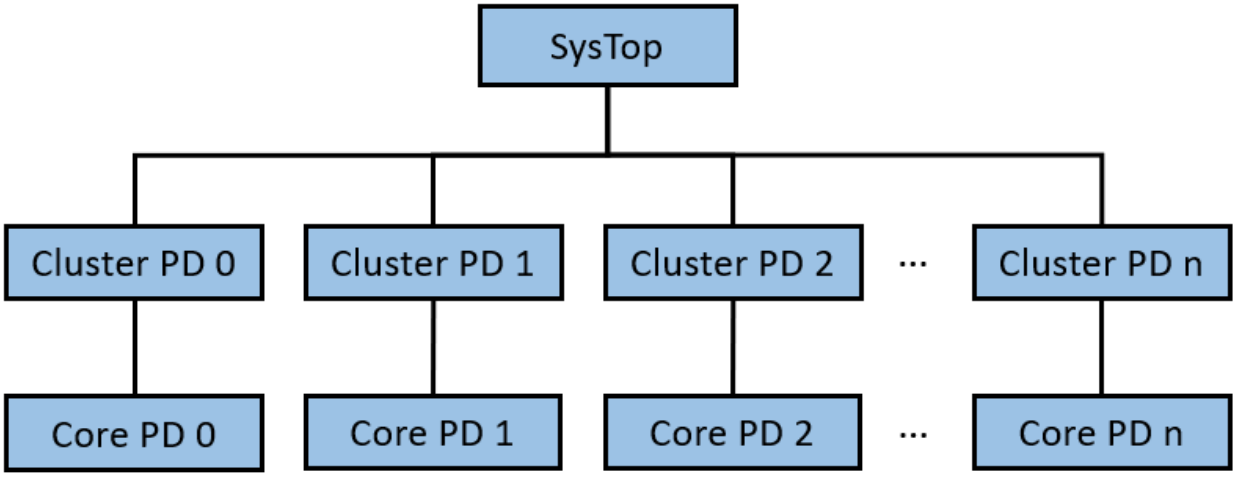
Subsequent to the switch to secure mode, CPU0 generates a Software Generated Interrupt (SGI) to bring the secondary CPUs into secure mode, triggering the execution of the ISR for the SGI. Within the ISR, secondary CPUs execute the power-down sequence, including disabling further interrupts and entering dynamic power-off.

Concurrently, while secondary cores are executing the ISR, the primary CPU dispatches an SCMI message to the System Control Processor (SCP), undertakes power-down activities, and enters dynamic power-off.

The distinction between shutdown, cold reboot and warm reboot occurs at the SCP.

Shutdown

The SCP manages a power tree for the RD-V3 platform, with SysTop power domain at the apex, followed by cluster power domains and CPU power domains at the lower levels.



To identify a power-down request, SCP analyses the SCMI message received from AP. The power-down process is executed in a bottom-to-top manner within the power tree. Starting from the bottom, SCP powers down the power domains sequentially (The current power domain HAL is simply returning success, this need to be updated to program the PPU registers).

Upon completing the power-down of power domains, SCP signals the RSE using the same SCMI message format received from AP. This prompts RSE to execute its shutdown sequence. Subsequently, SCP notifies the MCP with the identical SCMI messaging format.

Upon receiving the shutdown request, MCP initiates the power-down sequence and outputs the shutdown message to the UART console. This message serves as the trigger for the FVP to terminate its execution.

Cold reboot

The cold reboot flow mirrors the shutdown process until the communication with the RSE and MCP stages. In the case of a cold reboot, SCP alters the sequence by informing MCP before RSE.

Upon receiving the cold reboot message, MCP initiates its power-down sequence and enters WFI mode. RSE executes its power-down sequence and triggers a system-wide reset by programming the system reset register. This action induces a complete system reboot.

Warm reboot

During a warm reboot, the System Control Processor (SCP) undertakes the following steps:

1. SCP programs the Power Processing Units (PPUs) for all CPUs to transition into a static OFF state, while leaving the cluster PPUs untouched.
2. SCP then awaits the transition of all CPUs into the static OFF state.
3. After confirming that all cores have entered static OFF, SCP proceeds to power up the boot CPU.
4. The boot CPU starts from the BL1 stage, initiating the warm reboot process.

28.3.3 Download and build the required platform software

For downloading and building the platform firmware, refer *Busybox Boot*.

28.3.4 Validating Shutdown/Reboot

Shutdown

To verify the shutdown functionality, boot the platform to busybox. From busybox command line, issue the command

```
poweroff -f
```

Cold reboot

To verify the cold reboot functionality, boot the platform to busybox. From busybox command line, issue the command

```
reboot -f
```

Warm reboot

To verify the warm reboot functionality, boot the platform to Grub. Press the key 'e' to edit command line and append 'reboot=warm'. The resulting command line will appear as follows:

```
linux /Image acpi=force ip=dhcp root=PARTUUID=9c53a91b-e182-4ff1-aeac-
↪6ee2c432ae94 rootwait verbose debug reboot=warm
```

After making the edit, proceed with the boot by pressing the 'F10' key and allow the platform to boot into busybox. Once at the busybox command line, issue the following command:

```
reboot -f
```

28.4 System Monitoring Control Framework (SMCF)

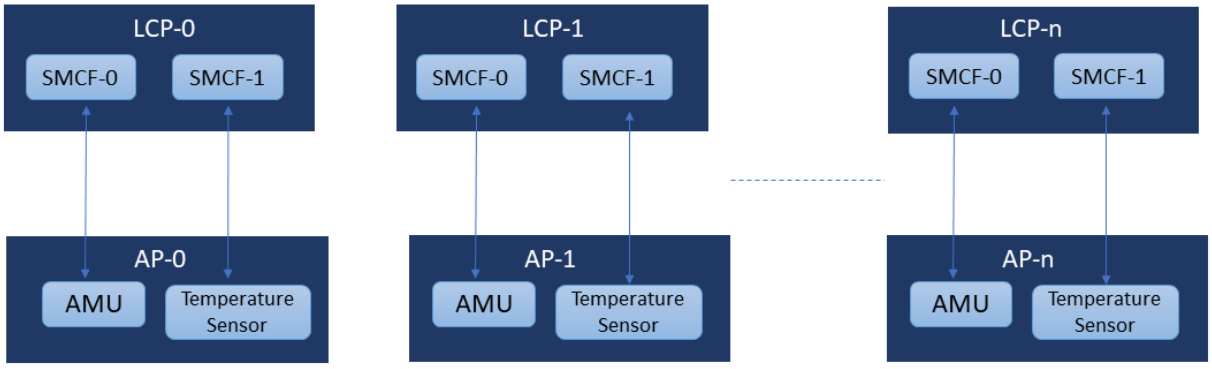
Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

28.4.1 Overview of SMCF

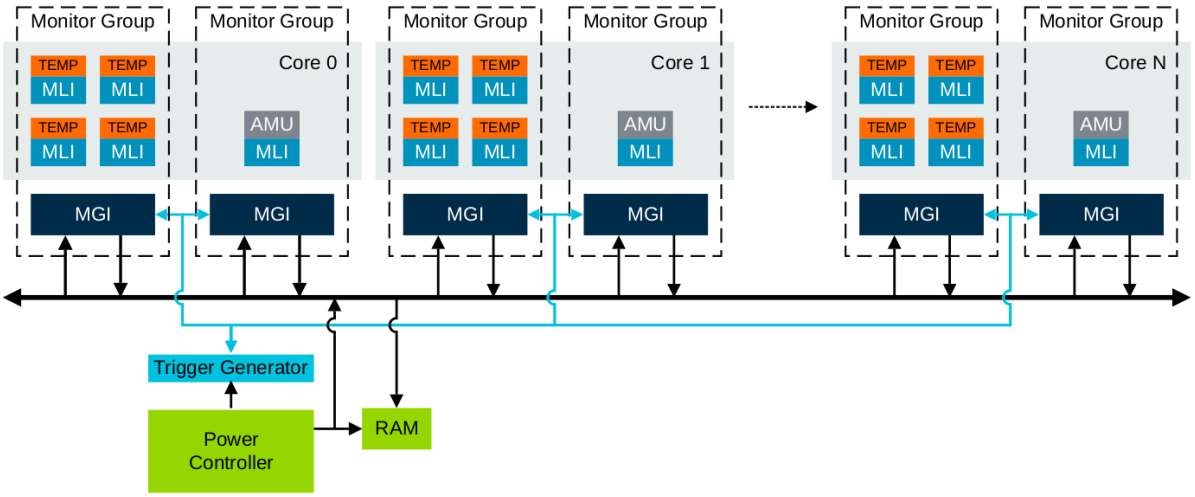
The System Monitoring Control Framework is designed to manage a large and diverse set of on-chip sensors and monitors. It does this by presenting software with a standard interface to control the monitors, regardless of type, and reducing software load of controlling the monitor sampling and data collection.

The SMCF reduces the burden on monitor control by enabling sampling on multiple monitors to be controlled together and by various triggers either internal or external to the SMCF. The number of monitors that the SMCF supports can be configured. The SMCF eases data collection requirements by allowing the data from multiple monitors to be collated in a single location or writing out data to a memory-mapped location that is easier for the monitoring agent to access.

SMCF can effectively manage sensors, track activity counters, and monitor dynamically evolving system data. The SMCF consists of two components, an MGI and an MLI.Each data source is called a monitor and connects to an MLI (Monitor Local Interface).The data width of each monitor could be anything from one bit to 64bits.Each group of MLI's is connected to one MGI (Monitor Group Interface),which provides the software interface and a set of functions to be applied to a group of monitors. SMCF MGIs (and related MLIs) are implemented in the LCP subsystem for core temperature sensors and AMU. The diagram below shows the SMCF internal view with LCP and AP:



There is a trigger input from the SCP, this is used to trigger a sample on the SMCF MGI. This allows the SCP to trigger a simultaneous sample on all relevant sensors and monitors.The single trigger input to the LCP is connected to all the MGI input triggers in the LCP. The diagram below gives the simplified SoC structure of SMCF:



There are four modes to sampling the data:

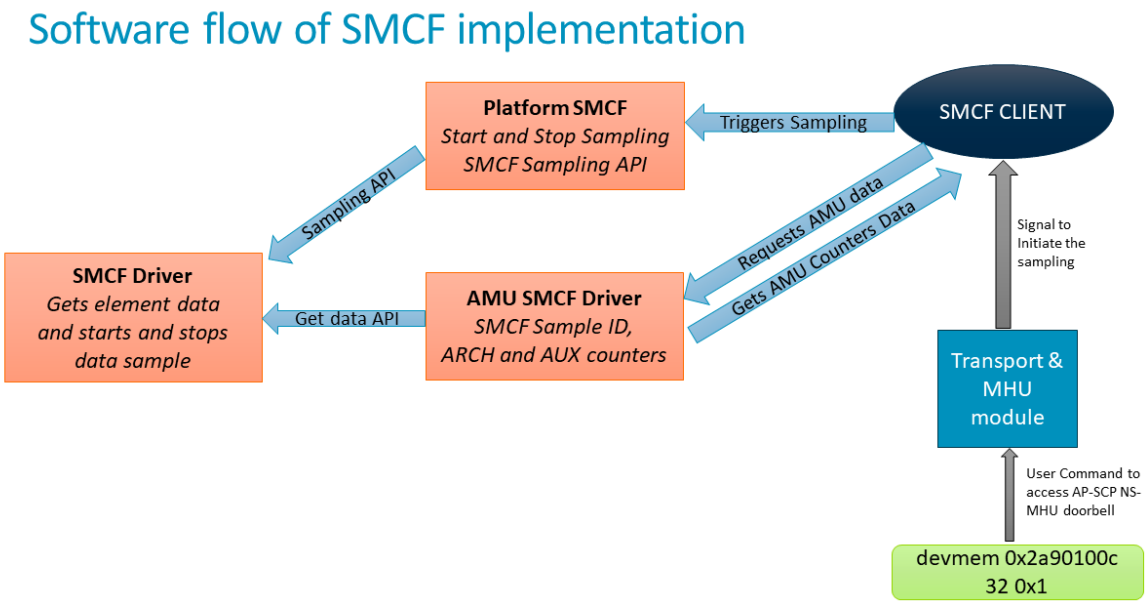
1. Manual Trigger : Initiated by the software for a single sample from the SMCF.
2. Periodic Sample: Software-driven continuous sampling at predefined interval.
3. Data Read: Data read sampling is used when a sample is required to be started when the data from the previous monitor sample data set is consumed. When the last data value from a monitor sample data set is read, a new sample begins.
4. Input Trigger: External event initiated sampling. Input trigger sampling is used when a sample is required to be started from an event that is external to an MGI.

28.4.2 SMCF Software Flow and Configuration

1. SCP accesses the SMCF Region through cluster utility mmap, which is mapped to the SCP address translation window.
2. The single trigger input to the LCP is connected to all the MGI input triggers in the LCP.Each MGI can be configured to start a sample based on this input trigger.
3. Software configures the MGI register base address,sample type, MGI write address,SMCF SRAM read address and respective IRQs.
4. Software is expected to write to this SMCF MGI Trigger enable register on a regular interval of time to initiate the sensor data collection. The trigger output from this register is expected to go to all MGIs.
5. The SMCF framework collect the data from MGI and update the SMCF SRAM on receiving the trigger. Software reads the sensor data from the SMCF SRAM.
6. Any platform with SMCF uses the SMCF to read out the AMU data instead of directly accessing the AMU data.
7. SMCF client module uses AMU smcf and platform smcf module for AMU data collection and for using the data sampling APIs.
8. The platform smcf module exposes platform specific data sampling APIs i.e start and stop sampling.

9. SMCF client module in SCP binds to AMU SMCF module to read out the AMU data, currently only the ARCH counters are being read.
10. SMCF client, on receiving instructions from the user, triggers the sampling and gives out AMU data as output in the console.
11. SMCF client is controlled by AP-SCP Non-secure MHU channel. SMCF client binds to Transport module for receiving MHU signal. User from AP Linux console rings AP-SCP Non-secure MHU channel doorbell. On receiving MHU interrupt MHU module through Transport module will signal SMCF client module to start, capture and stop SMCF sampling.

The diagram below explains the software flow of SMCF:



28.4.3 Download and build the required platform software

For downloading and building the platform firmware, refer *Busybox Boot* or *Buildroot boot* documentation.

28.4.4 Validating the SMCF

From the user end, start the SMCF sampling by following procedure:

1. Executing devmem command from Linux console for accessing AP-SCP NS-MHU doorbell channel.

```
devmem 0x2a90100c 32 0x1
```

2. This will launch the SMCF sampling and prints the collected sample data in the SCP console. The output will show 3 AMU counter values for all cores present in platform. For RD-V3-Cfg1 platform 8 such instances will be there. An example output looks like below:

```
[SMCF_CLIENT] Data successfully fetched for MGI[0]
[SMCF_CLIENT] MGI[0] AMU_COUNTER[0] data = 5077165163
[SMCF_CLIENT] MGI[0] AMU_COUNTER[1] data = 175172523
[SMCF_CLIENT] MGI[0] AMU_COUNTER[2] data = 5077165163
[SMCF_CLIENT] MGI[0] AMU_COUNTER[3] data = 0
```

(continues on next page)

(continued from previous page)

```
[SMCF_CLIENT] MGI[0] AMU_COUNTER[4] data = 0
[SMCF_CLIENT] MGI[0] AMU_COUNTER[5] data = 0
[SMCF_CLIENT] MGI[0] AMU_COUNTER[6] data = 0
[SMCF_CLIENT] Data successfully fetched for MGI[1]
[SMCF_CLIENT] MGI[1] AMU_COUNTER[0] data = 3201246
[SMCF_CLIENT] MGI[1] AMU_COUNTER[1] data = 110394
[SMCF_CLIENT] MGI[1] AMU_COUNTER[2] data = 3201246
[SMCF_CLIENT] MGI[1] AMU_COUNTER[3] data = 0
[SMCF_CLIENT] MGI[1] AMU_COUNTER[4] data = 0
[SMCF_CLIENT] MGI[1] AMU_COUNTER[5] data = 0
[SMCF_CLIENT] MGI[1] AMU_COUNTER[6] data = 0
```

28.4.5 Optional Changes for FVP based platforms

For getting precise readings on FVP, please use the parameters below: 1. Export these variables before launching the model:

```
export FASTSIM_DISABLE_TA=0
```

2. Pass `--quantum=400` as model parameter and pass `--min-sync-latency=1` also as model parameter.

CHAPTER

TWENTYNINE

RELIABILITY, AVAILABILITY, AND SERVICEABILITY (RAS)

29.1 Overview

Reliability, Availability and Serviceability (RAS) is a measure that defines the robustness of the system. A RAS enabled platform ensures that the system produces correct outputs, is always operational and is easily maintainable. RAS reduces the systems downtime by detecting the hardware errors and correcting them when possible. The level of RAS to be achieved is implementation dependent. There are various techniques that help achieve RAS targets e.g Fault prevention and fault removal, error handling and recovery and fault handling. A well designed RAS system ensures that the software and hardware collectively work to minimize the impact of hardware faults on entire system operation and hence boost performance.

RAS specification divides the entire RAS architectural extension support into two categories:

- ARMv8-A RAS Extension
- RAS System Architecture

RAS architectural specification defines the hardware RAS extensions that the cpu and the system could implement to achieve the desired level of RAS support.

ARMv8-A RAS Extension defines the RAS extensions that are mandatory for CPU implementation that are based on ARMv8.2 and above. To enable RAS extension architectural support in software the RAS_EXTENSION flag must be set to 1.

RAS system architecture defines the architectural support required to enable system level RAS support on a platform. It defines a reusable component architecture that can detect, record errors and also signal them to Processing Element (PE). PE is implementation defined, it can be anything that is capable of handling the given error e.g AP, SCP or MCP. This architectural definitions makes designing the software easier.

29.2 Component Definitions by RAS System Architecture

Below are some component definitions that the RAS System architecture defines:

29.2.1 Node

A node is one such component architecture defined by RAS. A system can have single or multiple error nodes. Architecturally a node:

- Implements one or more standard error record.
- Records detected and consumed errors.
- Might include control to disable the error reporting and recording while the software initializes.
- Reports recorded errors with asynchronous error reporting mechanism like interrupts e.g Fault Handling Interrupt (FHI).
- Implements a counter for counting corrected errors.
- Logs timestamps in each error record.
- Report uncorrected error by in-band error reporting signaling (external abort)
- Report critical error condition via Critical Error Interrupt (CRI).

29.2.2 Error Record

RAS system architecture defines standard error record. A node captures entire error information as part of these error records. Spec defines a mechanism to access error records as system register or memory mapped registers. A standard error record comprises of:

- ERR<n>STATUS: characterizes the error and marks valid status fields.
- ERR<n>ADDR: error address register.
- ERR<n>MISC<m>: miscellaneous error register. To be used for:
  - Identifying the Field Replaceable Unit (FRU).
  - Locating the error within the FRU.
  - Implementing corrected error counter to count the corrected errors.
  - Storing the timestamp value for recorded errors.

An Error record records following component error states:

- Corrected Error (CE).
- Deferred Error (DE).
- Uncorrected Error (UE): UE has following sub-types:
  - Uncontainable error (UC).
  - Unrecoverable error (UEU).
  - Recoverable error or Signaled error (UER).
  - Restartable error or Latent error (UEO).

29.3 Error Handling

There are two approaches to achieve error handling in software:

- *Firmware First Error Handling.*
- *Kernel First Error Handling.*

29.3.1 Firmware First Error Handling

Firmware First error handling requires the error events that occur are handled in EL3 and then relayed to OSPM for logging. On error firmware consumes the error information generates a standard Common Platform Error Record (CPER) information buffer which is defined by UEFI specification to store error information. CPER is placed in firmware reserved memory that is later shared with the OSPM when it is notified about the error.

On Arm Neoverse Reference design platforms the Firmware First error handling is achieved using Hardware Error Source Table (HEST) and Software Delegated Exception Interface (SDEI) tables. The Secure Partition (Standalone MM driver) is used to generate CPER info for the error. At boot the HEST table is published and OSPM is made aware about the hardware error source(s) the platform supports.

During the runtime when hardware fault is detected the corresponding error or fault handling interrupt is generated. This interrupt is taken to EL3 runtime firmware which calls into Secure Partition that generates CPER record and places it in firmware reserved memory. EL3 runtime firmware using SDEI notifies the OSPM about the error.

29.3.2 Kernel First Error Handling

Kernel First errors are handled directly by the OSPM without firmware intervention. The fault and error events that are generated by the platform are taken directly to OSPM.

Arm Neoverse Reference design platforms use Arm Error Source Table (AEST) to achieve kernel first error handling. AEST table is defined in ACPI specification for RAS specification. AEST table defines the hardware error sources that are present on the platform. AEST table comprises of one or more error nodes. A AEST node entry has information of component the node belongs to e.g Processor, Memory, SMMU, GIC etc. It defines interface type for accessing the node e.g memory mapped or system register. A node also defines the list of interrupts the node supports.

OSPM implements a AEST driver module to traverse through the AEST table. The module registers Irq handlers for all supported node interrupts. The fault event occurring on that node or error source is directly forwarded to OSPM for handling.

29.4 Error Injection

Error injection feature is a micro-architecture feature defined by RAS to inject errors in the RAS supported system components. Software can use these registers to inject the error and test the error handling software implemented by the platform.

Arm Neoverse Reference Designs use the Error Injection (EINJ) ACPI table defined in the ACPI specification to implement error injection feature. EINJ is action and instruction based table that defines set of actions and their corresponding instructions. Each action is also assigned a firmware reserved memory space to store action specific data. An instruction is essentially a read or a write operation that is performed on that reserved memory.

On Arm Neoverse Reference Platforms the firmware at EL3 implements the functionality to program the error injection registers. OSPM initiates the injection and generates an SPI interrupt to call in to firmware. EINJ defines a action to program the GICD register that triggers a SPI interrupt that is handled in EL3.

Firmware-first and Kernel-first software use the EINJ ACPI table to validate the software functionality.

Note: Error injection, whether firmware-first or kernel-first, are both initiated from the kernel.

29.4.1 Error Injection via Kernel

CPU Error Injection

The Neoverse RD-N2 platforms has support for 2 error nodes, and the presence of these nodes enable the RAS extension.

- Node 0: Includes the L3 memory system in the DSU.
- Node 1: Includes the private L1 and L2 memory systems in the cpu.

RD-V3 only supports one error node.

- Node 0: Includes the private L1 and L2 memory systems in the cpu.

CPU support SED parity (Single Error Detect) and SECDED ECC (Single Error Correct Double Error Detect) capabilities.

Rd-V3-Cfg1 and RdN2 platforms also supports injecting error's to verify error handling software.

Note: The Neoverse RD-V3 reference design platforms are based on direct connect configuration and has no DSU. Hence they only support one error node i.e Node0.

Error Injection Software Sequence

CPU implements Pseudo Fault Generation registers. With the help of these registers, software can inject either CE, DE or UE into the cache RAMs.

Detailed error injection software sequence:

- Select error record for L1 and L2 memory systems i.e. Node0
  - write_errselr_el1 (0)
- Program the Error Control Register to enable Error Detection, FHI for CE, DE and UE.
  - write_erxctlr_el1 (0x109) (Note: To enable ERI on UE write 0x10D)
- Program the PFG Control Register to 0.
  - write_cpu_pfg_ctrl_register (0)
- Clear the Error Status Register to 0.
  - write_erxstatus_el1 (0xFFC00000)
- Set PFG countdown register to 1.
  - write_cpu_pfg_cdn_register (1)
- For Deferred Error injection write
  - write_cpu_pfg_ctrl_register (0x80000020) [Generates FHI interrupt]

Procedure to Perform Error Injection

Note: This section assumes the user has completed the *Getting Started* chapter and has a functional working environment.

Error Handling Mode Selection

CPU supports both *Firmware First* and *Kernel First* error handling modes, and the default mode is set to *Firmware First*.

Important: Only one error handling mode can be enabled at a time.

The error handling modes are a build time option, in order to select either the user needs to navigate to the `<workspace>` and edit the configuration file of the platform of interest and look for `TF_A_RAS_FW_FIRST` flag.

As an example for RD-V3 Cfg1 platform:

```
vim <workspace>/build-scripts/configs/rdv3cfg1/rdv3cfg1
```

- Firmware First Selection:

```
TF_A_RAS_FW_FIRST = 1
```

- Kernel First Selection

```
TF_A_RAS_FW_FIRST = 0
```

Note: Clean and build once you switch error handling mode.

Build and Boot Operating System(s)

Refer to any of the bellow list of supported operating systems, to build the reference design platform software stack and boot into the OS.

- *Busybox Boot*
- *Buildroot Boot*

Inject Error

After the boot is complete, based on the error handling scheme selected use EINJ table debugfs entries to inject the error.

- *Firmware First Error Injection.*
- *Kernel First Error Injection.*

The field `sel-firmware-first` in oem-einj is used to toggle firmware first error injection, with the default being kernel first error injection. Field `sel-error-type` is used to choose the type of error injection, where the current implementation only support's deferred errors.

Firmware First Error Injection

```
mount -t debugfs none /sys/kernel/debug # Step needed for Buildroot only
echo 0x80020000 > /sys/kernel/debug/apei/einj/error_type
echo 1 > /sys/kernel/debug/apei/einj/oem-einj/sel-firmware-first
echo 2 > /sys/kernel/debug/apei/einj/oem-einj/sel-component
echo 2 > /sys/kernel/debug/apei/einj/oem-einj/sel-error-type
echo 1 > /sys/kernel/debug/apei/einj/error_inject
```

On successful error injection the firmware reception log's this error information on the console.

Check the secure uart terminal (window with the name `FVP terminal_sec_uart`) for a log similar to below.

```
SP 8001: ErrAddr = 0x8F840
SP 8001: MmEntryPoint Done
INFO:    EINJ event received 83
INFO:    cpu_id 2
INFO:    Injecting DE...
INFO:    ErrStatus = 0x0
INFO:    [CPU RAS] CPU intr received = 17 on cpu_id = 2
INFO:    [CPU RAS] ERXMISC0_EL1 = 0x0
INFO:    [CPU RAS] ERXSTATUS_EL1 = 0x40800000
INFO:    [CPU RAS] ERXADDR_EL1 = 0x0 buff_base = 0xf4600000
```

Check the non-secure uart terminal (window with the name `FVP terminal_nsec_ uart`) for a log similar to below.

```
{2}[Hardware Error]: Hardware error from APEI Generic Hardware Error Source: 10
{2}[Hardware Error]: event severity: recoverable
{2}[Hardware Error]:  Error 0, type: recoverable
{2}[Hardware Error]:   section_type: ARM processor error
{2}[Hardware Error]:   MIDR: 0x00000000410fd840
{2}[Hardware Error]:   Multiprocessor Affinity Register (MPIDR): 0x0000000081020000
{2}[Hardware Error]:   running state: 0x1
{2}[Hardware Error]:   Power State Coordination Interface state: 0
{2}[Hardware Error]:   Error info structure 0:
{2}[Hardware Error]:   num errors: 1
{2}[Hardware Error]:    first error captured
{2}[Hardware Error]:    error_type: 0, cache error
{2}[Hardware Error]:    error_info: 0x000000000002001f
{2}[Hardware Error]:     transaction type: Generic
{2}[Hardware Error]:     operation type: Generic error (type cannot be determined)
{2}[Hardware Error]:     cache level: 0
{2}[Hardware Error]:     processor context not corrupted
{2}[Hardware Error]:     the error has not been corrected
{2}[Hardware Error]:    physical fault address: 0x0000000000000000
{2}[Hardware Error]:   Context info structure 0:
{2}[Hardware Error]:    register context type: AArch64 general purpose registers
{2}[Hardware Error]:    00000000: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000010: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000020: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000030: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000040: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000050: 00000000 00000000 00000000 00000000
```

(continues on next page)

(continued from previous page)

```
{2}[Hardware Error]:    00000060: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000070: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000080: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    00000090: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000a0: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000b0: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000c0: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000d0: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000e0: 00000000 00000000 00000000 00000000
{2}[Hardware Error]:    000000f0: 00000000 00000000 00000000 00000000
```

Kernel First Error Injection

```
mount -t debugfs none /sys/kernel/debug # Step needed for Buildroot only
echo 0x80020000 > /sys/kernel/debug/apei/einj/error_type
echo 0 > /sys/kernel/debug/apei/einj/oem-einj/sel-firmware-first
echo 2 > /sys/kernel/debug/apei/einj/oem-einj/sel-component
echo 2 > /sys/kernel/debug/apei/einj/oem-einj/sel-error-type
echo 1 > /sys/kernel/debug/apei/einj/error_inject
```

On successful error injection the kernel receives a error event which is received in the irq handler. The handler traverses through the error record info and logs the error.

Check the non-secure uart terminal (window with the name `FVP terminal_nsec_ uart`) for a log similar to below.

```
[ 2365.760926] Injecting DE-
[ 2365.760928] ARM RAS: error from CPU7
[ 2365.760930] ERR0STATUS: 0x40800000
```

EDAC (Error Detection and Correction)

The EDAC(Eror Detection and Correction) Linux interface provides a framework, for reporting memory and CPU errors encountered on a system. It allow the Kernel to detect and manage errors, providing valuable information for diagnostics and troubleshooting hardware issue.

We currently only enabled EDAC support for CPU for both RD-N2 and RD-V3 Platforms. Error count is exposed through sysfs inteface this interface allows user to access information about Corrected (CE) and Uncorrected (UE) errors that have occurred in the system aiding in monitoring and diagnosing hardware issues.

Note: This feature is only supported on RD-V3-Cfg1 and RD-N2-Cfg1 Platforms if Kernel first error Handling is enabled.

```
cat /sys/devices/system/edac/cpu/cpu*/ue_count
```

Shared RAM Error Injection

RD-V3 and RD-N2 platform have support for Shared RAM that is shared between AP, MCP, SCP and RSS. The shared RAM is protected with SECDED (Single Error Correct Double Error Detect). RD-V3 platform defines ECC RAS registers to log any ECC errors that occur during Shared RAM access from each master AP, SCP, MCP or RSS. For RD-V3 4 sets of ECC RAS registers defined for each master to log errors based on master's PAS and 2 sets of ECC Ras registers for RD-N2 platform.

RD-V3: The list for Shared RAM ECC RAS registers is defined below:

- AP Secure RAM ECC RAS registers
- AP Non-Secure RAM ECC RAS registers
- AP Realm RAM ECC RAS registers
- AP Root RAM ECC RAS registers
- SCP Secure RAM ECC RAS registers
- SCP Non-Secure RAM ECC RAS registers
- SCP Realm RAM ECC RAS registers
- SCP Root RAM ECC RAS registers
- MCP Secure RAM ECC RAS registers
- MCP Non-Secure RAM ECC RAS registers
- MCP Realm RAM ECC RAS registers
- MCP Root RAM ECC RAS registers

RD-N2: The list for Shared RAM ECC RAS registers is defined below:

- AP Secure RAM ECC RAS registers
- AP Non-Secure RAM ECC RAS registers
- SCP Secure RAM ECC RAS registers
- SCP Non-Secure RAM ECC RAS registers
- MCP Secure RAM ECC RAS registers
- MCP Non-Secure RAM ECC RAS registers

Note: This test is only supported on RD-V3-Cfg1 and RD-N2-Cfg1 Platforms. *Firmware First Error Handling*

Error Injection on Shared RAM

Each ECC RAS register set implements SRAMECC_ERRMISC1 register which provides a way to inject Corrected Error (CE) or Uncorrected Error (UE) in the Shared RAM. The error injection only takes effect if the register programming is followed by a read access to shared RAM. If the injection is successful the error records pertaining to the master and respective access are populated with error information and an error interrupt is delivered to the master.

RD-V3 Shared SRAM

Detailed Error injection software sequence is illustrated to inject 1-bit CE into Root Shared RAM from AP executing in RD-V3.

- Add memory map for the Shared RAM ECC RAS registers memory space.
- Add memory map for the Shared memory space.
- Program the SRAMECC_ERRCTRL register to enable ED(Error detection), FI(Fault Interrupt) and CFI(Corrected Fault Interrupt)
- Program the SRAMECC_ERRMISC1 register to enable INJECT_CE.
- Read from memory mapped shared memory space to inject the error.

RD-N2 Shared SRAM

Detailed Error injection software sequence is illustrated to inject 1-bit CE into Non-Secure Shared RAM from AP executing in RD-N2.

- Add memory map for the Shared RAM ECC RAS registers memory space.
- Add memory map for the Shared memory space.
- Program the SRAMECC_ERRCTRL register to enable RAM_ECC_EN and set INJECT_ERROR to [01] for Correctable error.
- Read from memory mapped shared memory space to inject the error.

Procedure to Perform Error Injection on Shared RAM

Note: This section assumes the user has completed the *Getting Started* chapter and has a functional working environment.

Error Handling Mode Selection

Both platform only supports *Firmware First* SRAM error handling mode, and the default mode is set to *Firmware First*.

Important: Only Firmware first mode is supported for SRAM-Errors.

The error handling modes are a build time option, in order to select either the user needs to navigate to the `<workspace>` and edit the configuration file of the platform of interest and look for `TF_A_RAS_FW_FIRST` flag.

As an example for RD-V3 Cfg1 platform:

```
vim <workspace>/build-scripts/configs/rdv3cfg1/rdv3cfg1
```

- Firmware First Selection:

```
TF_A_RAS_FW_FIRST = 1
```

Note: Clean and build once you switch error handling mode.

Build and Boot Operating System(s)

Refer to any of the bellow list of supported operating systems, to build the reference design platform software stack and boot into the OS.

- *Busybox Boot*
- *Buuidroot Boot*

Inject Error on Shared RAM

Run below command to inject 1-bit CE to the Shared RAM. This test uses EINJ ACPI table to perform error injection. Shared RAM is not a standard defined error_type in EINJ ACPI table so use the vendor defined error type. Bit 31 of error_type field represents vendor error type. Use error_type value 0x8002_0000 to represent Shared RAM errors.

```
mount -t debugfs none /sys/kernel/debug # Step needed for Buildroot only
echo 0x80020000 > /sys/kernel/debug/apei/einj/error_type
echo 1 > /sys/kernel/debug/apei/einj/oem-einj/sel-firmware-first
echo 1 > /sys/kernel/debug/apei/einj/oem-einj/sel-component
echo 1 > /sys/kernel/debug/apei/einj/oem-einj/sel-error-type
echo 1 > /sys/kernel/debug/apei/einj/error_inject
```

Shared RAM error handling happens in Firmware first mode. The EL3 firmware receives the fault handling interrupt (FHI) for the corrected error detected and logs the error on the secure console.

```
EDAC MC0: 1 CE unknown error on unknown memory
( page:0x8f offset:0x840 grain:-281474976710655 syndrome:0x0 - APEI location: )
{1}[Hardware Error]: Hardware error from APEI Generic Hardware Error Source: 20
{1}[Hardware Error]: It has been corrected by h/w and requires no further action
{1}[Hardware Error]: event severity: corrected
{1}[Hardware Error]:  Error 0, type: corrected
{1}[Hardware Error]:   section_type: memory error
{1}[Hardware Error]:   physical_address: 0x000000000008f840
{1}[Hardware Error]:   physical_address_mask: 0x0000ffffffffffff
```

29.4.2 Error Injection via SCP Utility

The error injection utility is referred to as einj-util in this document. Einj-util is a command-line utility designed for SCP. This utility integrates with the SCP CLI Debugger, enabling users to insert commands at runtime. Einj-util facilitates error injection into various RAS-supported components when a user provides error injection command input in the CLI. This utility helps in validating the RAS capable hardware components' behavior when error is detected and reported.

The term "Component" defines the RAS-supported components for which error injection is supported. "Sub-component" signifies the next level of error categorization for each component, and it varies for different components. For instance, in the context of SRAM, sub-components represent error injection in different worlds: Root, Secure, Realm, and Non-Secure. "Type" defines the various types of errors supported by each component. Error types supported are Correctable Error(CE), Deferred Error(DE), Uncorrectable Error(UE).

Procedure to Perform Error Injection into Various Components

Note: This section assumes the user has completed the *Getting Started* chapter and has a functional working environment.

Build Software Stack

This procedure doesn't require a full host OS to be present, but the *Busybox Boot* is still recommended as it is the simplest method to build the required components.

Boot up to SCP CLI Debugger Shell

Once the build step is completed, boot the Busybox stack on FVP as normal but identify the window with the name `FVP terminal_uart_scp` once it shows up, as this window is the one to interact with. The steps are as follows:

- Launch the FVP and access the SCP UART.
- Once in the SCP UART terminal, use `Ctrl + e` to enter the CLI.
- To access the help menu for the einj-util utility, run the command

```
einj-util -h
```

- The "help" command displays the CLI usage.

```
> einj-util -h
    Inject error into various components.

    Usage: einj-util -comp <n> -subcomp <n> -type <n>

    -comp: sram (0), tcm (1), cpu (2), rsm (3)

    -subcomp:

            sram: root (0), secure (1), non-secure (2), realm (3)

            tcm: itcm (0), dtcm (1)

            cpu: always 0 for now

            rsm: secure (0), non-secure (1)

    -type:

            sram/tcm/rsm: correctable (0), uncorrectable (1)

            cpu: correctable (0), uncorrectable (1), deferred (2)

    example:

            1) ce into shared sram from secure world:
```

(continues on next page)

(continued from previous page)

```
                einj-util -comp 0 -subcomp 1 -type 0
        2) ce into scp itcm:
                einj-util -comp 1 -subcomp 0 -type 0
        3) cpu ue:
                einj-util -comp 2 -subcomp 0 -type 1
```

- To exit the CLI Debugger, press `Ctrl + d`.

Various Error Injection Scenarios

Component	Subcomponent	Type of Error	Error Status
Shared SRAM	Secure World	CE	0x86000000
	Root World	UE	0xa4000000
RSM SRAM	Secure World	CE	0x86000000
	Non-Secure World	UE	0xa4000000
TCM	ITCM	CE	0x5
	DTCM	UE	0x7
CPU	Core	CE	0xC6000000
		UE	0x60000000
		DE	0x40800000

Shared SRAM Error Injection

Run the following command to inject a correctable error into shared SRAM from the secure world.

```
> einj-util -comp 0 -subcomp 1 -type 0
```

After triggering the error, the interrupt handler is invoked, logging error records.

```
[SRAM_INT] ErrStatus = 0x86000000
[SRAM_INT] fwk_int number = 24
[SRAM_INT] ErrAddr   = 0x10
```

SRAM ECC Error Status Register Bit Descriptions

```
AV[31:31]  :  Address Valid

MV[26:26]  :  Miscellaneous Registers Valid

CE[25:24]  :  Correctable error has occurred

DE[23:23]  :  Deferred Error

UET[21:20] :  Uncorrected Error Type

SERR[7:0]  :  Primary Error code
```

CPU Error Injection

Run the following command to inject a CPU correctable error.

```
> einj-util -comp 2 -subcomp 0 -type 0
```

The ErrorStatus register captures information about the triggered CPU error.

```
Injecting CPU CE
ErrStatus  0xC6000000
ErrAddress 0x0
```

Core Error Injection ERXSTATUS_EL1 Register Description

```
AV[31:31]  :  Address Valid

V[30:30]   :  Status Register Valid

MV[26:26]  :  Miscellaneous Registers Valid

CE[25:24]  :  Corrected Error

DE[24:24]  :  Deferred Error

UET[21:20] :  Uncorrected Error Type

SERR[4:0]  :  Primary Error code
```

SCP ITCM/DTCM Error Injection

Invoke the following command to inject a correctable error into SCP ITCM.

```
> einj-util -comp 1 -subcomp 0 -type 0
```

The error record information will be logged in the following manner.

```
ITCM
Injecting CE
[TCM_INT] fwk_int number = 21
[TCM_INT] ErrCode   = 0x9
[TCM_INT] ErrStatus = 0x5
[TCM_INT] ErrAddr   = 0x34d8
```

TCMECC_ERRSTATUS Bit Descriptions

```
OF[2:2] : Multiple errors occurred before SW cleared the current error

UE[1:1] : Uncorrectable and uncontainable error have occurred

CE[0:0] : Correctable error has occurred
```

RSM SRAM Error Injection

Invoke the following command to trigger a correctable error in RSM SRAM from the secure world.

```
> einj-util -comp 3 -subcomp 0 -type 0
```

The error record information is logged as follows:

```
Injecting CE into RSM SRAM
[RSM_INT] ErrStatus = 0x86000000
[RSM_INT] fwk_int number = 29
[RSM_INT] ErrAddr   = 0x10
```

Note: Refer to the SRAM ECC Error Status register bit descriptions to decode the error status for RSM SRAM errors.

Expected Output for the Various Scenarios

Description	Command	Expected Output
Shared SRAM Secure World CE	*einj-util -comp 0 -subcomp 1 -type 0*	Injecting CE into Shared SRAM [SRAM_INT] ErrStatus = 0x86000000 [SRAM_INT] fwk_int number = 24 [SRAM_INT] ErrAddr = 0x10
Shared SRAM Secure World UE	*einj-util -comp 0 -subcomp 1 -type 1*	Injecting UE into Shared SRAM [SRAM_INT] ErrStatus = 0xa4000000 [SRAM_INT] fwk_int number = 24 [SRAM_INT] ErrAddr = 0x1
Shared SRAM Root CE	*einj-util -comp 0 -subcomp 0 -type 0*	Injecting CE into Shared SRAM [SRAM_INT] ErrStatus = 0x86000000 [SRAM_INT] fwk_int number = 26 [SRAM_INT] ErrAddr = 0x10
Shared SRAM Root UE	*einj-util -comp 0 -subcomp 0 -type 1*	Injecting UE into Shared SRAM [SRAM_INT] ErrStatus = 0xa4000000 [SRAM_INT] fwk_int number = 26 [SRAM_INT] ErrAddr = 0x10
RSM SRAM Secure World CE	*einj-util -comp 3 -subcomp 0 -type 0*	Injecting CE into RSM SRAM [RSM_INT] ErrStatus = 0x86000000 [RSM_INT] fwk_int number = 29 [RSM_INT] ErrAddr = 0x10
RSM SRAM Secure World UE	*einj-util -comp 3 -subcomp 0 -type 1*	Injecting UE into RSM SRAM [RSM_INT] ErrStatus = 0xa4000000 [RSM_INT] fwk_int number = 29 [RSM_INT] ErrAddr = 0x10
RSM SRAM Non-secure World CE	*einj-util -comp 3 -subcomp 1 -type 0*	Injecting CE into RSM SRAM [RSM_INT] ErrStatus = 0x86000000 [RSM_INT] fwk_int number =

29.5 Rasdaemon

29.5.1 Overview

Rasdaemon is error logging tool that is used to log RAS (Reliability, Availability and Serviceability) events. The daemon uses the kernel trace sub-system to capture the error events reported by the kernel modules. The trace events that are captured in /sys/kernel/debug/tracing are reported by the rasdaemon.

Enabling rasdaemon creates a "instances/rasdaemon" directory inside "/sys/kernel/debug/tracing" debugfs directory. All the tracing events that are enabled by the rasdaemon are captured in this directory.

Note: This test is only supported on RD-V3-Cfg1 and RD-N2-Cfg1 Platforms. *Firmware First Error Handling*

29.5.2 Enabling Rasdaemon

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

RD-N2-Cfg1 and RD-V3-Cfg1 platform have rasdaemon package enabled by default on the buildroot file system. Buildroot repository has support added to enable rasdaemon, any platform performing a buildroot boot can enable rasdaemon package.

To enable rasdaemon on other platform variants add following code to the buildroot defconfig file.

```
BR2_PACKAGE_RASDAEMON=y
BR2_GLOBAL_PATCH_DIR="board/aarch64-efi/rdinfra/patches/"
```

To add rasdaemon support on RD-V3 platform add above two lines to file configs/rdv3/buildroot/aarch64_rdinfra_defconfig

Build the software stack for buildroot. Refer *Build the platform software*.

Perform buildroot filesystem boot. Refer *Booting with Buildroot as the filesystem*.

On the buildroot shell type following command to enable rasdaemon

```
mount -t debugfs none /sys/kernel/debug
rasdaemon -e
```

This command starts rasdaemon and enables trace events for memory controller, aer, non_standard error records, arm event and arm ras external events.

```
rasdaemon: ras:mc_event event enabled
rasdaemon: ras:aer_event event enabled
rasdaemon: ras:non_standard_event event enabled
rasdaemon: ras:arm_ras_ext_event event enabled
rasdaemon: ras:arm_event event enabled
```

29.5.3 Test to validate rasdaemon

To validate the logging of RAS events by rasdaemon requires a platform with RAS support enabled. Here we look at the 1-bit DE reported by the CPU on RD-V3-Cfg1 platform that has RAS support enabled. Perform the test for firmware first error handling for 1-bit DE on CPU. The kernel logs this event and also reports an arm_event for this error to the tracing subsystem. Rasdaemon captures this arm_event trace log and prints it.

Refer *CPU Error Injecton* to perform CPU firmware first error handling test on RD-V3-Cfg1 platform. On the error injection the kernel logs the error and also the arm_event. The trace event is also recorded as part of rasdaemon buffer. To log the trace from rasdaemon run following command.

```
cat /sys/kernel/debug/tracing/instances/rasdaemon/trace_pipe
```

The above command outputs following log from rasdaemon.

```
<idle>-0       [004] d.h1.   555.977157: arm_event: affinity level: 255;
MPIDR: 0000000081040000; MIDR: 00000000410fd840; running state: 1; PSCI state: 0
```

29.5.4 Other components supporting RAS

29.6 CMN Cyprus Kernel First Handling (KFH)

Important: This feature might not be applicable to all Platforms. Please check section **Supported Features** of individual platform pages to confirm if this feature is listed as supported. Also this feature can be validated only on a pre-silicon validation platform. Current support is limited to RASv1.

29.6.1 CMN Cyprus RAS support

CMN Cyprus implements RAS as a distributed architecture with set of logging, reporting registers and a central interrupt handling unit. The logging and reporting registers are implemented in the XP, HN-I, HN-F/S, SBSX and CCG device nodes.

Logging registers implemented in the device node are:

- Error Feature register (ErrFr)
- Error Control register (ErrCtlr)
- Error Status register (ErrStatus)
- Error Address register (ErrAddr)
- Error Misc register 0 (ErrMisc0)
- Error Misc register 1 (ErrMisc1)

Two sets of these registers are implemented by each device node, one to log error that occur when in root address space and other to log the error when executing in non-secure address space. Each device node also implements ErrGsr (Error group status register) that is set when that node is has non-zero ErrStatus register. CMN Cyprus supports following error types:

- Corrected Error (CE)
- Deferred Error (DE)

- Uncorrected Error Unrecoverable (UEU)

Example: In RD-V3-Cfg1 platform implements a CMN mesh of size 3*3. That has 9 XP's, 8 HNS, 1 SBSX, 4 HN-I and 5 CCG device nodes. Each of these nodes implement a set of error records to log the detected RAS errors.

Each device node also implements the Pseudo Fault Generation (PFG) registers that allows to inject the pseudo errors within the device node and validate the software error handling flow. The PFG registers defined for each node are:

- Error Pseudo Fault Generation Feature register (ErrPfgf)
- Error Pseudo Fault Generation Control register (ErrPfgctl)
- Error Pseudo Fault Generation Count Down register (ErrPfgcdn)

There are 2 sets of PFG registers implemented per device node. One for root world error injection and other for NS world error injection.

29.6.2 Error/Fault injection in CMN Cyprus

Sequence to be followed to perform SW induced error injection:

- Program the Error Control register to enable error detection and enable the FHI interrupt

```
mmio_write_errctlr ((CMN_BASE + NODE_OFF + ErrCtlr), (BIT3 | BIT0))
```

- Program the PFG count down register to 1, to inject error on first clock tick.

```
mmio_write_pfgcdn ((CMN_BASE + NODE_OFF + ErrPfgCdn), 1)
```

- Program the PFG control register with following fields:
  - Type of error, if CE set BIT6, if DE set BIT5, if UEU set BIT2
  - Set BIT11 to update ErrStatus.AV field on fault injection
  - Set BIT12 to update ErrStatus.MV field on fault injection
  - Set BIT31 to enable the injection by reading the PFG count down register

```
mmio_write_pfgCtlr ((CMN_BASE + NODE_OFF + PfgCtlr), (BIT<Error_Type> |
BIT11 | BIT12 | BIT31))
```

Run this same sequence in order to inject the error in any of the CMN device node. NODE_OFF for each node must be known before performing the injection, which can be determined from the CMN discovery process.

29.6.3 CMN KFH Software

To enable CMN KFH following SW components are required.

- Arm Error Source Table (AEST) ACPI table to represent CMN errors
- SSDT table
- AEST device driver for CMN.

SSDT Table

Add one entry in the SSDT table to define the CMN cyprus device memory CRS object. Refer ACPI for Arm Components spec for more information on various field details.

```
// CMN 800 device
 Device (CMN8) { // CMN-800 device object for an X * Y
   Name (_HID, "ARMHC800")
   Name (_UID, Zero)
   Name (_CRS, ResourceTemplate () {
     // Descriptor for 1 GB of the CFG region at offset PERIPHBASE
     QWordMemory (
       ResourceConsumer,
       PosDecode,
       MinFixed,
       MaxFixed,
       NonCacheable,
       ReadWrite,
       0x00000000,       // Granularity
       0x100000000,      // Min, set to PERIPHBASE
       0x13FFFFFFF,      // Max
       0x000000000,      // Translation
       0x040000000,      // Range Length 1GB
       ,                 // ResourceSourceIndex
       ,                 // ResourceSource
       CFGM              // DescriptorName
     )
   })
 } // Device(CMN8)
```

AEST table

Each RAS capable device node is represented as AEST node within the AEST table. E.g below is the AEST node entry for HNF0, where 0 represent the logical ID of the HNF. For more information refer ACPI for the RAS and ACPI for Arm Components specs. These specs describes all the necessary fields to be populated to define a AEST node for a given CMN device node.

```
{
  .NodeResource = {
    .Vendor = {
      {
        EFI_ACPI_AEST_NODE_TYPE_VENDOR_DEFINED,    /* Type */
        sizeof (EFI_ACPI_AEST_NODE_DATA),          /* Length */
        0,                                         /* Reserved */
        sizeof (EFI_ACPI_AEST_NODE_STRUCT),        /* Offset to Node data */
        sizeof (EFI_ACPI_AEST_NODE_RESOURCE),      /* Offset to Node Interface */
        (sizeof (EFI_ACPI_AEST_NODE_RESOURCE) +    /* Offset to Node Interrupt */
         sizeof (EFI_ACPI_AEST_INTERFACE_STRUCT)),
        1,                                         /* Interrupt array size */
        0,                                         /* Timestamp */
        0,                                         /* Reserved1 */
        0,                                         /* Injection countdown rate */
```

(continues on next page)

(continued from previous page)

```
      },
      // Vendor Node Structure
      AEST_NODE_TYPE_VENDOR_HID,                    /* Hardware ID */
      1,                                            /* Unique ID */
      // Vendor Data
      {
        0x00,                                       /* Offset HNF0 0x1700000 */
        0x00,
        0x70,
        0x01,
        0,
        0,
        0,
        0,
        0x00,                                       /* Offset HND 0x0000 */
        0x00,
        0,
        0,
      },
    },
  },
  {
    EFI_ACPI_AEST_INTERFACE_TYPE_MMIO,        /* Interface type */
    {0, 0, 0},                                /* Reserved */
    0,                                        /* Flags */
    0,                                        /* Base Address */
    0,                                        /* Record Index */
    0,                                        /* Num Error records */
    0,                                        /* Record implemented */
    0,                                        /* Group status reporting */
    0,                                        /* Addressing mode */
    0,                                        /* ACPI ARM error node device */
    0,                                        /* Processor Affinity */
    0,                                        /* ErrGsr base address */
  },
  {
   {
      EFI_ACPI_AEST_INTERRUPT_TYPE_FAULT_HANDLING,     /* Interrupt type */
      {0, 0},                                          /* Reserved */
      EFI_ACPI_AEST_INTERRUPT_FLAG_TRIGGER_TYPE_LEVEL, /* Flags */
      79,                                              /* GSIV */
      0,                                               /* ID */
      {0, 0, 0},                                       /* Reserved */
    },
  },
},
```

Note that HNF0 error node does not define anything in the interface structure. CMN relies completely on the Vendor-defined nodedata structure to communicate the device node offset and respective HND node offset.

AEST CMN driver for CMN

The AEST driver for CMN is implemented as an extension to the AEST ACPI table driver. The AEST CMN driver at boot reads the SSDT table and reads the CRS object to determine the CMN base address and size and creates virtual mapping the CMN address space.

Each CMN device error node data is represented using the vendor-defined structure in the AEST ACPI table. At boot the AEST ACPI driver parses the AEST table and when it locates a vendor node, it adds the node data to a platform device structure and registers a platform device. AEST ACPI driver registers a platform device driver to process the vendor defined errors. For each AEST node of type vendor error that is detected by the AEST ACPI driver it registers a platform device and calls into the probe function. For each platform device registered if the vendor HID is set to CMN HID, it is registered with the AEST CMN driver.

The AEST CMN driver reads the vendor platform device information into a driver specific data structure. The AEST CMN driver maintains the device structure in the linked list. Each list entry holds the information for all the error nodes of same device type. Driver also registers the IRQ handlers to process the FHI interrupt generated when a device node detects CE, DE or UE. On an error event the IRQ handler parses through all the device node structures and reads the ErrGsr register for each node. For a non-zero ErrGsr located the handler logs the error records, clears the interrupt and returns. Below is a example log for DE detected on HNS0 and HNI1

```
[    2.117375] AEST_CMN: RAS v2 enabled = 0
[    2.118373] AEST_CMN: Error record registers for device node HNS0
[    2.119858] AEST_CMN: [HNS0] ErrFr_NS = 0x5200008012c9a2
[    2.121154] AEST_CMN: [HNS0] ErrCtlr_NS = 0x10d
[    2.122263] AEST_CMN: [HNS0] ErrStatus_NS = 0xc4800000
[    2.123512] AEST_CMN: [HNS0] ErrAddr_NS = 0x0
[    2.124573] AEST_CMN: [HNS0] ErrMisc0_NS = 0x0
[    2.125656] AEST_CMN: [HNS0] ErrMisc1_NS = 0x0
[    2.140341] AEST_CMN: RAS v2 enabled = 0
[    2.141305] AEST_CMN: Error record registers for device node HNI1
[    2.142784] AEST_CMN: [HNI1] ErrFr_NS = 0x120000801201a2
[    2.144077] AEST_CMN: [HNI1] ErrCtlr_NS = 0x10d
[    2.145181] AEST_CMN: [HNI1] ErrStatus_NS = 0xc4800000
[    2.146430] AEST_CMN: [HNI1] ErrAddr_NS = 0x0
[    2.147491] AEST_CMN: [HNI1] ErrMisc0_NS = 0x0
[    2.148570] AEST_CMN: [HNI1] ErrMisc1_NS = 0x0
```

CHAPTER

THIRTY

SYSTEMREADY COMPLIANCE PROGRAM

Arm SystemReady is a compliance program that helps ensure the interoperability of an operating system on Arm-based hardware. Developers can build software once and deploy it on any compliant Arm-based chip. Arm SystemReady benefits the entire ecosystem as existing software can move through hardware generations and hardware providers can reach a broader addressable market.

Arm SystemReady is implemented in alignment with the two OS environment and firmware approaches commonly used today. These are split into the following bands:

- SystemReady Band
- SystemReady Devicetree Band

Neoverse Reference Designs focus on the SystemReady Band only. For more information, visit: Arm SystemReady Compliance Program.

30.1 SystemReady Band

Arm SystemReady band is part of the Arm SystemReady Compliance Program. It helps ensure operating system interoperability for advanced configuration and power interface (ACPI) environments where generic operating systems (OS) can be installed on either new or old hardware without modification. Old OSs can run on new hardware, and new OSs can run on old hardware, without customization.

Arm SystemReady band-compliant hardware helps reduce the cost of supporting multiple versions of operating systems through seamless interoperability with standard operating systems, hypervisors, and software.

Arm SystemReady band is relevant for systems using Windows, Linux, VMware, and BSD environments.

Arm SystemReady band compliant systems must conform to the following:

- Base System Architecture (BSA) specification
- Server Base System Architecture (SBSA) supplement specification
- SBBR recipe of the Arm Base Boot Requirements (BBR) Specification

Arm SystemReady band compliance and testing requirements are specified in the Arm SystemReady Requirements Specification (SRS).

30.2 System Architecture Compliance Suites (ACS)

SystemReady band is a subset of System Architecture Compliance Suites which are a series of test suites that check the compliance of a system against arm architectural specifications.

This chapter describes how to build Neoverse Reference Designs software stack and execute the test-suites that achieve compliance against the specifications.

It is recommended to use prebuilt systemready-images as shown bellow, but for reference on the sources and instructions to build from source refer to SystemReady band Github repository.

30.2.1 Build the Platform Software

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

Note:

- This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.
- Example provided here is for RDN2 Platform
- The target `all` executes the individual `clean`, `build` and `package` in sequence, if calling `build` alone due to an incremental build, it must be followed by `package`.

The components of the RD platform software stack that are built are limited to those that provide the EFI implementation and the EFI shell (i.e, SCP, TF-A and EDK2).

Command to build the software stack is as follows:

```
./build-scripts/rdinfra/build-test-acs.sh -p rdn2 all
```

30.2.2 Prepare Test Image

The prebuilt systemready-image is generic for the different SystemReady Bands, so an extra step is required to execute the SBSA test specification in automation mode.

Prebuilt system-ready images are available to download from Github in prebuilt_images. After the download is complete, extract the image using:

```
unzip systemready_acs_live_image.img.xz.zip
xz -d systemready_acs_live_image.img.xz
```

Inspect the image to check the offset needed to mount the partition:

```
sudo fdisk -l systemready_acs_live_image.img.xz
```

The output should be similar to:

```
Disk systemready_acs_live_image.img: 642 MiB, 673185792 bytes, 1314816 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 6F4BF28C-FC58-45A7-9614-B52782B7F71B

Device                           Start     End Sectors  Size Type
systemready_acs_live_image.img1   2048 1312766 1310719  640M Microsoft basic data
```

The partition starts at sector *2048*, so we need to multiply the *sector size* with the *start sector*, thus `512*2048=1048576`

Mount the image in the filesystem to edit `acs_run_config.ini` file and change the property `SbsaRunEnabled` from `0` to `1`:

```
mkdir sr-image
sudo mount -t vfat systemready_acs_live_image.img sr-image -o loop,offset=1048576,uid=
↪$(id -u),gid=$(id -g)
vim sr-image/acs_tests/config/acs_run_config.ini
```

30.2.3 Execute Test Image

Note: A new copy of the disk image shall be used every time the tests are to be executed, this ensures tests are not skipped due to presence of log files from the previous executions.

Supported command line options are listed below:

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -v <systemready_acs_live_image.img>
  - The absolute path to the systemready_acs_live_image.img has to be supplied as the parameter.
- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Execute RD-N2 platform with networking enabled and the systemready-image located at `/tmp/systemready_acs_live_image.img`.

```
export MODEL=<absolute path to the platform FVP binary>
cd model-scripts/rdinfra
./acs.sh -p rdn2 -v /tmp/systemready_acs_live_image.img -n true
```

The SBSA/SBBR tests are split into two phases - tests that execute from linux and the tests that execute from an EFI interface level.

Let the boot progress to the Grub menu. To execute SystemReady band tests, choose the option `SystemReady band ACS (Automation)` from Grub menu, which launches the EFI shell.

Press `Enter` key or wait until the timeout in the EFI shell to finish.

The systemready-image by default executes SBBR SCT tests. To skip this suite, wait until `Press any key to stop the EFI SCT running` displays in the log and press any key. If SBBR SCT is not skipped, the SBSA will execute after completion of SBBR SCT.

On SBSA test completion, the script reboots the platform, follow the steps mentioned above until skipping SCT.

This time SBSA is not executed as the results are already captured in the systemready_acs_live_image.img image and the validation OS starts to boot.

The Linux part of the test will be executed on validation OS boot complete.

On completion of SBSA and SBBR tests, the execution stops at the command line prompt. The execution can be stopped by terminating the FVP.

In case it is not required to run the complete ACS compliance, i.e.: validate only the SBSA, the systemready_acs_live_image.img has the provision for this. SBSA test should be run from the EFI shell manually by executing the command listed below:

```
Shell> EFI\BOOT\bsa\sbsa\Sbsa.efi -skip 800
```

Running the test manually will not store the test result into systemready_acs_live_image.img disk image, instead the test results will be available on the console as the test proceeds to completion.

30.2.4 Retrieve Test Results

On completion of SBSA/SBBR tests, test results can be retrieved by mounting the partition of systemready-image that was used for the test. The offset is calculated the same as shown above, thus execute:

```
mkdir sr-image
sudo mount -t vfat systemready_acs_live_image.img sr-image -o loop,offset=1048576,uid=
↪$(id -u),gid=$(id -g)
```

The test results can be found in the directories below:

- UEFI SBSA test report: sr-image/acs_results/uefi/
- Linux SBSA test report: sr-image/acs_results/linux/
- FWTS result: sr-image/acs_results/fwts/

Unmount the image after analysing the logs using the following command:

```
sudo umount sr-image
```

30.2.5 Select a SBSA Compliance Level (Optional)

SBSA specification classifies hardware into different levels, level-3 through level-7. The systemready-image defaults to level 4. To select a different level press ESC from UEFI shell and run the SBSA efi binary manually to select the appropriate compliance level to be tested. An example command to select the compliance level is:

```
Shell> EFI\BOOT\bsa\sbsa\Sbsa.efi -l <y>

# Where, y can be 3 to 7 for the SBSA compliance level.
```

CHAPTER

THIRTYONE

TF-A TESTS

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

31.1 Overview of tf-a-tests

The Trusted Firmware-A Tests (TF-A-Tests) is a suite of baremetal tests to exercise the Trusted Firmware-A (TF-A) features from the Normal World. Neoverse Reference Design (RD) platform software stack supports booting TF-A-Tests. This enables strong TF-A functional testing without dependency on a Rich OS. Refer the Trusted Firmware-A Tests Documentation for more details.

This document describes how to build the Neoverse RD platform software stack and and use it to boot TF-A-Tests on the Neoverse RD FVP.

31.2 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

This section describes the procedure to build the platform firmware required to boot TF-A-Tests on Neoverse RD platforms.

To build the software stack, the command to be used is

```
./build-scripts/build-test-tf-a-tests.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`
    - `build`

    * package
    * all (all of the three above)

Note: On networks where git port is blocked, the build procedure might not progress. Refer the *troubleshooting guide* for possible ways to resolve this issue.

Examples of the build command are

- Command to clean, build and package the RD-N2 software stack required for TF-A-Tests boot on RD-N2 platform:

```
./build-scripts/build-test-tf-a-tests.sh -p rdn2 all
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform.

```
./build-scripts/build-test-tf-a-tests.sh -p rdn2 build
```

Note: This command should be followed by the package command to complete the preparation of the FIP.

- Command to package the previously built software stack and prepare the FIP.

```
./build-scripts/build-test-tf-a-tests.sh -p rdn2 package
```

31.3 Boot TF-A-Tests

After the build of the platform software stack for TF-A-Tests is complete, the following commands can be used to start the execution of the *selected platform fastmodel* and boot the TF-A-Tests. Examples on how to use the command are listed below.

To boot TF-A-Tests, the commands to be used are

- Set MODEL path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Launch TF-A-Tests boot:

```
./tftf.sh -p <platform name> -a <additional_params> -n [true|false]
```

Supported command line options are listed below

- -p <platform name>
    - Lookup for a platform name in Platform Names.

- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to boot TF-A-Tests are as listed below.

- Command to start the execution of the RD-N2 model to boot TF-A-Tests:

```
./tftf.sh -p rdn2
```

- Command to start the execution of the RD-N2 model to boot TF-A-Tests with network enabled. The model supports virtio.net allowing the software running within the model to access the network:

```
./tftf.sh -p rdn2 -n true
```

- Command to start the execution of the RD-N2 model with networking enabled and to boot TF-A-Tests. Additional parameters to the model are supplied using the -a command line parameter:

```
./tftf.sh -p rdn2 -n true -a "-C board.flash0.diagnostics=1"
```

- Once the tests complete, a message similar to the following output will be displayed on the non-secure UART terminal. This demonstrates the usage of TF-A Tests on Arm infrastructure reference design platforms.

```
********************************* Summary *********************************
> Test suite 'Framework Validation'
                                                                    Passed
> Test suite 'Timer framework Validation'
                                                                    Passed
> Test suite 'Boot requirement tests'
                                                                    Passed
> Test suite 'Query runtime services'
                                                                    Passed
> Test suite 'PSCI Version'
                                                                    Passed
> Test suite 'PSCI Affinity Info'
                                                                    Passed
> Test suite 'CPU Hotplug'
                                                                    Passed
> Test suite 'PSCI CPU Suspend'
                                                                    Passed
> Test suite 'PSCI STAT'
                                                                    Passed
> Test suite 'PSCI NODE_HW_STATE'
                                                                    Passed
> Test suite 'PSCI Features'
                                                                    Passed
> Test suite 'PSCI MIGRATE_INFO_TYPE'
                                                                    Passed
> Test suite 'PSCI mem_protect_check'
                                                                    Passed
```

(continues on next page)

(continued from previous page)

```
> Test suite 'SDEI'
                                                                    Passed
> Test suite 'Runtime Instrumentation Validation'
                                                                    Passed
> Test suite 'TRNG'
                                                                    Passed
> Test suite 'IRQ support in TSP'
                                                                    Passed
> Test suite 'TSP handler standard functions result test'
                                                                    Passed
> Test suite 'Stress test TSP functionality'
                                                                    Passed
> Test suite 'TSP PSTATE test'
                                                                    Passed
> Test suite 'EL3 power state parser validation'
                                                                    Passed
> Test suite 'State switch'
                                                                    Passed
> Test suite 'CPU extensions'
                                                                    Passed
> Test suite 'ARM_ARCH_SVC'
                                                                    Passed
> Test suite 'Performance tests'
                                                                    Passed
> Test suite 'SMC calling convention'
                                                                    Passed
> Test suite 'FF-A Setup and Discovery'
                                                                    Passed
> Test suite 'SP exceptions'
                                                                    Passed
> Test suite 'FF-A Direct messaging'
                                                                    Passed
> Test suite 'FF-A Power management'
                                                                    Passed
> Test suite 'FF-A Memory Sharing'
                                                                    Passed
> Test suite 'SIMD,SVE Registers context'
                                                                    Passed
> Test suite 'FF-A Interrupt'
                                                                    Passed
> Test suite 'SMMUv3 tests'
                                                                    Passed
> Test suite 'FF-A Notifications'
                                                                    Passed
> Test suite 'PMU Leakage'
                                                                    Passed
> Test suite 'DebugFS'
                                                                    Passed
> Test suite 'Realm payload tests'
                                                                    Passed
================================
Tests Skipped : 122
```

(continues on next page)

(continued from previous page)

```
Tests Passed  : 60
Tests Failed  : 0
Tests Crashed : 0
Total tests   : 182
================================
NOTICE:  Exiting tests.
```

CHAPTER

THIRTYTWO

UEFI SELF-CERTIFICATION TEST

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

32.1 Overview of SCT Standalone test

The UEFI Self-Certification Test (UEFI SCT) is a toolset for platform developers to validate firmware implementation compliance to the UEFI Specification. The toolset features a Test Harness for executing built-in UEFI Compliance Tests, as well as for integrating user-defined tests that were developed using the UEFI SCT open source code.

The latest version of the UEFI SCT can be found at the UEFI website

This document describes how to build the Neoverse RD platform software stack and and use it to run UEFI SCT on the Neoverse RD FVP.

32.2 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

This section describes the procedure to build the disk image for SCT run. The disk image consists of two partitions. The first partition is a EFI partition and contains grub. The second partition is a ext3 partition which contains the linux kernel image. Examples on how to use the build command for SCT are listed below.

To build the software stack, the command to be used is

```
./build-scripts/rdinfra/build-test-sct.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`

    * build
    * package
    * all (all of the three above)

Examples of the build command are

- Command to clean, build and package the RD-N2 software stack required for SCT on RD-N2 platform:

```
./build-scripts/rdinfra/build-test-sct.sh -p rdn2 all
```

- Command to perform an incremental build of the software components included in the software stack for the RD-N2 platform.

```
./build-scripts/rdinfra/build-test-sct.sh -p rdn2 build
```

Note: This command should be followed by the package command to complete the preparation of the UEFI SCT disk image.

- Command to package the previously built software stack and prepare the SCT disk image.

```
./build-scripts/rdinfra/build-test-sct.sh -p rdn2 package
```

32.3 Run UEFI SCT

After the build of the platform software stack for SCT is complete, the following commands can be used to start the execution of the *selected platform fastmodel* and run UEFI SCT. Examples on how to use the command are listed below.

To run UEFI sct, the commands to be used are

- Set MODEL path before launching the model:

```
export MODEL=<absolute path to the platform FVP binary>
```

- If platform is SGI-575:

```
cd model-scripts/sgi
```

- If platform is an RD:

```
cd model-scripts/rdinfra
```

- Run UEFI SCT:

```
./sct.sh -p <platform name> -a <additional_params> -n [true|false]
```

Supported command line options are listed below

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -j [true|false]
  - Automate SCT: true or false. Default value is set to 'false'.

- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example commands to run UEFI SCT are as listed below.

- Command to start the execution of the RD-N2 model to run UEFI SCT:

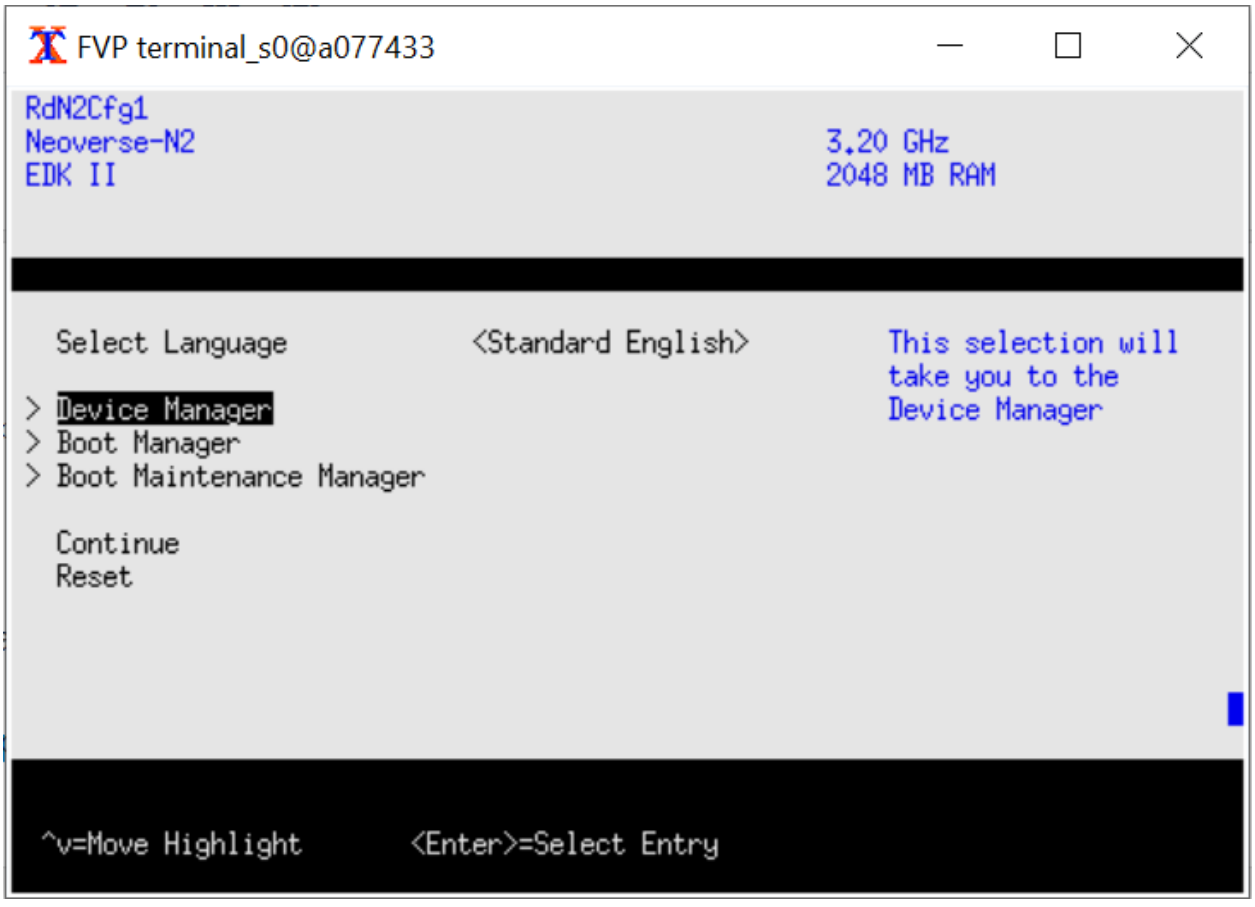
```
./sct.sh -p rdn2
```

- Command to start the execution of the RD-N2 model to run UEFI SCT with network enabled. The model supports virtio.net allowing the software running within the model to access the network:

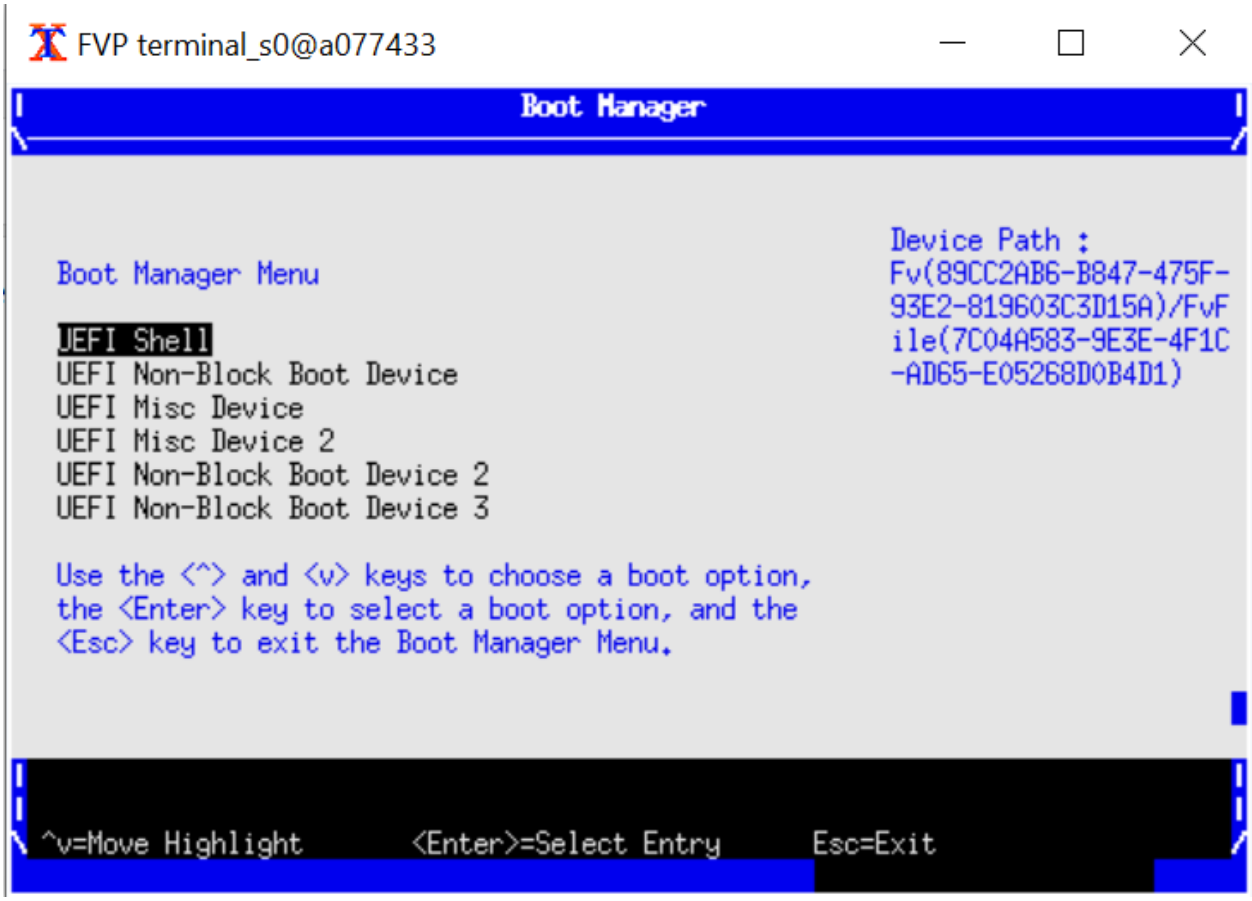
```
./sct.sh -p rdn2 -n true
```

- There are additional steps to be performed on the first boot to run SCT test. These steps are listed below.

1- Click ESC and click on "Boot Manager"

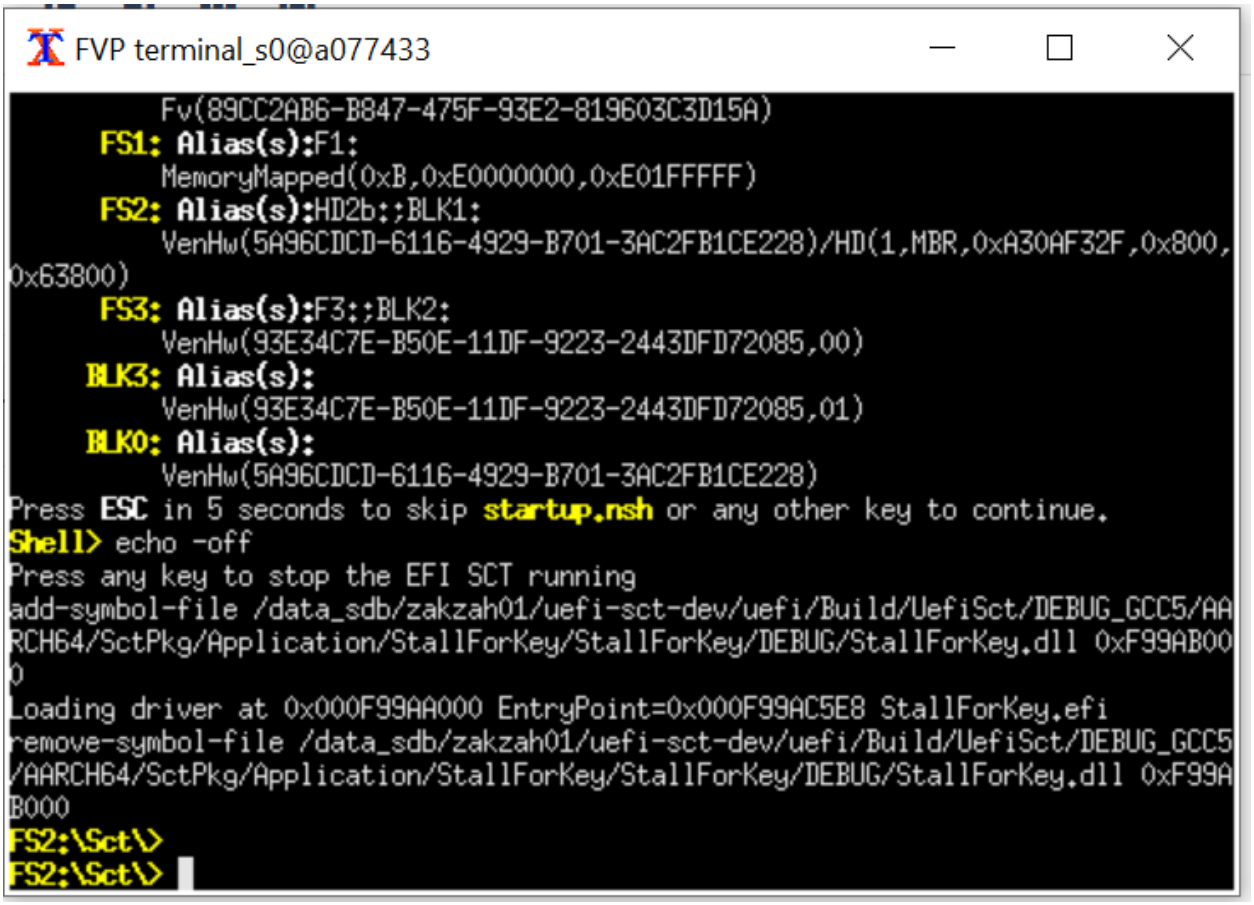


2- Select UEFI Shell and click Enter



3- In the UEFI Shell tap the command to run the SCT test navigator

```
SCT.efi -u
```



4- Select “Test Case Management”. Then, select any test you want to run. - To select a test tap “Space”, [1] should be printed in #Iter - To deselect a test tap again “Space”, [0] should be printed in #Iter

5- Click on F9 to run the selected tests

6- Retrieve the test results in the “View Test Log…”


FVP terminal_s0@a077433
UEFI2.7 Self Certification Test(SCT2)
Main Menu
Description
> Test Case Management
> Test Environment Configuration
> Test Device Configuration
> View Test Log ...
> Test Report Generator ...
View test log
F4/Dn Reset results F6 Save Sequence ESCer Exitct SubMenu

CHAPTER

THIRTYTHREE

VIRTUALIZATION

33.1 Virtualization using KVM

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.1.1 What is KVM?

Kernel Virtual Machine (KVM) is a virtualization module built in the Linux kernel which lets the user to turn Linux into a hypervisor to allow hosting single/multiple isolated guests or virtual machine. In brief, KVM is a type-2 hypervisor that requires a host OS to boot first, and the KVM module runs on top of that.

KVM requires a processor with hardware virtualization extensions. Some of the architectural features in Arm v8-a profile that support hardware virtualization are -

- A dedicated Exception level (EL2) for hypervisor code.
- Support for trapping exceptions that change the core context or state.
- Support for routing exceptions and virtual interrupts.
- Two-stage memory translation, and
- A dedicated exception for Hypervisor Call (HVC).

Currently, KVM is part of Linux kernel. Some of the features of KVM are:

- Over-committing: KVM allows to allocate more virtualized CPU or memory for the virtual machine than that of the host.
- Thin provisioning: KVM allows to allocate and optimize the flexible storage for the virtual machines.
- Disk throttling: KVM allows to set limits for disk I/O requests.
- Virtual CPU hot plug: KVM allows ability to increase the CPU count of the virtual machine during run time.

33.1.2 Virtualization on Neoverse Reference Design Platforms

Virtualization using KVM hypervisor is supported on the Neoverse reference design plaforms. The subsequent sections below provide detailed instructions about booting up of two or more instances of guest OS's (or Virtual Machines, VMs) using lkvm tool. Each of these guests can support upto NR_CPUS as vcpus, where NR_CPUS is the number of CPUs booted up by the host OS. There are instructions on using hardware virtualization features on the platform and enable use of virtualized devices, such as console, net, and disk etc.

33.1.3 Overview of Native Linux KVM tool

kvmtool is a lightweight tool for hosting KVM guests. As a pure virtualization tool it only supports guests using the same architecture, though it supports running 32-bit guests on those 64-bit architectures that allow this.

The kvmtool supports a range of arm64 architectural features such as support for GIC-v2, v3, and ITS. It also supports device virtualization using emulated devices such as virtio device support for console, net, and disks, and using VFIO to allow PCI pass-through or direct device assignment.

33.1.4 Booting multiple guests

Virtualization using KVM hypervisor requires a root filesystem from which kvmtool can be launched. Buildroot root filesystem supports the kvmtool package. It fetches the mainline kvmtool source and builds the kvmtool binary out of it. Detailed description on buildroot based booting ia available in *Buildroot guide*. Follow all the instructions in that document for building the platform software stack and booting upto buildroot before proceeding with the next steps.

To boot two or more virtual machines on the host kernel with a kernel image and an initrd or a disk image, KVMtool virtual machine manager (VMM) (also called as lkvm tool) is used. Check help for 'lkvm run' command for options to launch guests.

Launching multiple guests using lkvm:

- Mount grub disk-image: The buildroot filesystem required to perform kvm test is packaged in such a way that the kernel image, and buildroot ramdisk image are copied to the second partition of grub disk image that gets probed at /dev/vda2 in the host kernel. After booting the platform this partition can be mounted as:

  ```
  mount /dev/vda2 /mnt
  ```

- Launch VMs using lkvm: For launching multiple VMs, 'screen' tool can be used to multiplex console outputs so that one can switch between multiple workspaces. This tool helps by providing a new console output pane for each guest. Use the following command to launch guests using kvmtool with the available kernel and ramdisk images.

  ```
  screen -md -S "<screen_name>" /mnt/kvmtool/lkvm run -k <path-to-linux-image>
  ↪ -i <path-to-ramdisk-image> --irqchip gicv3-its -c <nr-cpus> -m <allowed-
  ↪mem> --console serial --params "console=ttyS0 --earlycon=uart,mmio,
  ↪0x1000000 root=/dev/vda"
  ```

  For example, to run the kernel available in mounted disk at /mnt as above use the following command:

  ```
  screen -md -S "virt1" /mnt/kvmtool/lkvm run -k /mnt/Image -i /mnt/ramdisk-buildroot.
  ↪img --irqchip gicv3-its -c 4 -m 512 --console serial --params "console=ttyS0 --
  ↪earlycon=uart,mmio,0x1000000 root=/dev/vda"
  ```

  Above command uses an emulated UART device by passing the argument '–console serial'. To use virtio based console (prints a bit faster than the emulated UART device) use the below command.

```
screen -md -S "virt1" /mnt/kvmtool/lkvm run -k /mnt/Image -i /mnt/ramdisk-buildroot.
↪img --irqchip gicv3-its -c 4 -m 512 --console virtio --params "earltprintk=shm␣
↪console=hvc0 root=/dev/vda"
```

- Launch couple of more guests by repeating the above command and updating the screen_name.

  The launched screens can be viewed from the target by using the following command:

```
screen -ls
```

- Select and switch to the desired screen to view boot-up logs from guest. Use the following command to go to a specific screen:

```
screen -r <screen_name>
```

  - For example, list of screens are shown below:

```
# screen -ls
There are screens on:
    214.virt1       (Detached)
    200.virt2       (Detached)
```

  - Jump to the screen using:

```
screen -r virt1
```

  - Switch between multiple running guests using 'Ctrl-a d' to view the bootup logs of various guests executing.

- Perform simple cpu hotplug test to validate that guest kernel is functional. Use the following command to do that:

```
echo 0 > /sys/devices/system/cpu/cpu1/online
echo 0 > /sys/devices/system/cpu/cpu2/online

echo 1 > /sys/devices/system/cpu/cpu1/online
echo 1 > /sys/devices/system/cpu/cpu2/online
```

  The CPUs should go offline and come back online with the above set of commands.

- Jump back to the host by exiting the screen using 'Ctrl-a d', and use the following command to see how many guests are managed by lkvm tool:

```
# /mnt/kvmtool/lkvm list
PID NAME                 STATE
------------------------------------
309 guest-309            running
276 guest-276            running
```

- Power-off the guests by jumping to the right screen and executing the command:

```
poweroff
```

- The guests would shutdown and the following message would be displayed on the console.

```
# KVM session ended normally.
```

This completes the procedure to launch multiple VMs and terminate them.

33.2 KVM Unit Tests

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.2.1 Overview of kvm-unit-tests

KVM unit testing started off alongside of original project KVM. It's purpose was to validate all the supported features of *KVM*. With evolving development of KVM, very quickly it became necessary to standardize the process of validation which motivated the *kvm-unit-tests* project. It's basically a collection of small standalone programs which are used as tiny guest operating systems (OS) to test KVM. Since KVM is part of Linux kernel, any userspace virtual machine manager (VMM) such as, *qemu* or *kvmtool* can be used for launching these guest OSes which then validate specific feature as implemented by the Linux KVM. Running of these unit testcases will also help in validation of the userspace hypervisor tool. The *kvm-unit-tests* framework supports multiple architectures e.g. i386, x86_64, armv7 (arm), armv8 (arm64), ppc64, ppc64le, and s390x. To learn more about the framework and testdevs please follow the official page for kvm-unit-tests.

33.2.2 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

This section describes the procedure to build the sofware stack required to perform KVM unit testing. Following software packages from the Neoverse reference platform software stack are needed to do the testing:

- Software stack for buildroot boot as given in *Buildroot* guide,
- kvm-unit-tests built for kvmtool target,
- Kvmtool VMM.

Skip this section if a *Buildroot* build is already completed for the platform as the `kvmtool` and `kvm-unit-tests` are already built as part of the buildroot build of the platform software stack.

- To build the software stack for buildroot boot, the command to be used is

```
./build-scripts/rdinfra/build-test-buildroot.sh -p <platform name> <command>
```

  Supported command line options are listed below

  - <platform name>
    - Lookup a platform name in Platform Names.
  - <command>
    - Supported commands are
      - `clean`
      - `build`
      - `package`
      - `all` (all of the three above)

Examples of the build command are:

- Command to clean, build and package the software stack needed for the buildroot boot on RD-N2 platform:

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2 all
```

33.2.3 Booting the platform for validation

- Boot the target platform FVP with Buildroot filesystem up to the buildroot prompt.
  - Set `MODEL` path before launching the model:

```
cd model-scripts/rdinfra
export MODEL=<absolute path to the platform FVP binary>
```

  - Launch buildroot boot:

```
./boot-buildroot.sh -p <platform name> -a <additional_params> -n [true|false]
```

  - For example, to start the execution of the RD-N2 model to boot up to the buildroot prompt with network enabled:

```
./boot-buildroot.sh -p rdn2 -n true
```

Running Unit Testcases

The kvmtool (lkvm) and kvm-unit-tests binaries are available in the filesystem on the second partition of the virtio-disk. Mount the vda2 partition on /mnt

```
mount /dev/vda2 /mnt
ls
  Image                 lost+found            test_smmute.sh
  kvm-ut                ramdisk-buildroot.img
  kvmtool               smmute
```

- Navigate to the kvm-unit-tests directory and launch the tests script to start all kvm-unit-tests. The test script uses LKVM environment variable with the correct path to lkvm binary.

```
cd kvm-ut
LKVM=/mnt/kvmtool/lkvm ./run_tests_kvmtool_arm.sh
```

- This launches all kvm-unit-tests on the platform. Output would look as below:

```
=== selftest-setup ===

PASS: selftest: setup: smp: number of CPUs matches expectation
INFO: selftest: setup: smp: found 2 CPUs
PASS: selftest: setup: mem: memory size matches expectation
INFO: selftest: setup: mem: found 256 MB
SUMMARY: 2 tests

EXIT: STATUS=1
```

(continues on next page)

(continued from previous page)

```
   === selftest-vectors-kernel ===

PASS: selftest: vectors-kernel: und
PASS: selftest: vectors-kernel: svc
PASS: selftest: vectors-kernel: pabt
SUMMARY: 3 tests

EXIT: STATUS=1

   === selftest-vectors-user ===

PASS: selftest: vectors-user: und
PASS: selftest: vectors-user: svc
SUMMARY: 2 tests

EXIT: STATUS=1

   === selftest-smp ===

INFO: selftest: smp: PSCI version: 1.1
INFO: selftest: smp: PSCI method: hvc
INFO: selftest: smp: CPU  1: MPIDR=0080000001
INFO: selftest: smp: CPU  2: MPIDR=0080000002
INFO: selftest: smp: CPU  3: MPIDR=0080000003
INFO: selftest: smp: CPU  4: MPIDR=0080000004
INFO: selftest: smp: CPU  5: MPIDR=0080000005
....
....
....
```

This completes the procedure to run kvm-unit-tests for KVM based virtualization validation on the platform.

33.3 Using non-discoverable devices connected to I/O virtualization block

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.3.1 Overview

The reference design platforms that support a IO Virtualization block as part of the compute subsystem allow connecting non-discoverable devices (non-PCIe) to it. The I/O virtualization block includes SMMUv3 to translate address and provide device isolation security, GIC-ITS to support MSI interrupts and NI-700 inter-connect to route transactions in and out of x16, x8, x4_1, and x4_0 ports.

The non-discoverable devices that are connected to the Io Virtualization block include - two PL011 UART, two PL330 DMA controllers and six regions of SRAM memory. The reference design software stack includes support to configure and use these devices and memory regions.

This document describes how to build the Neoverse reference design platform software stack and use it to test non-PCI devices that are connected on I/O virtualization blocks. *Busybox Boot* is used on the Neoverse RD FVP and tests are run from the command line to validate the devices.

NOTE: These tests are supported only on reference design platforms that designate one of the IO virtualization block for connecting non-discoverable devices.

33.3.2 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

33.3.3 Running tests for non-PCI devices on busybox

To begin the tests with the non-discoverable devices connected to the IO Virtualization block, boot to busybox using the command mentioned below (refer to *Busybox Boot* guide for details on the parameters).

```
./boot.sh -p <platform name> -a <additional_params> -n [true|false]
```

PL011 UART

There are two PL011 UART controllers connected to the non-discoverable IO Virtualization block. These UART controllers are initialized by edk2 firmware before booting to the Linux kernel.

These UART peripherals are enumerated by the Linux Kernel as Serial ports and can be tested by writing to the corresponding tty device.

- After booting into busybox, verify that the two PL011 UART controllers are enumerated. The command provided below will list the serial ports detected by the Linux kernel.

```
# dmesg | grep tty
[    0.034995] ARMH0011:00: ttyAMA0 at MMIO 0x1080000000 (irq = 14, base_baud = 0)␣
↪is a SBSA
[    0.035195] ARMH0011:01: ttyAMA1 at MMIO 0x10a0000000 (irq = 15, base_baud = 0)␣
↪is a SBSA
[    0.035595] ARMH0011:02: ttyAMA2 at MMIO 0xef70000 (irq = 36, base_baud = 0) is␣
↪a SBSA
[    0.037095] printk: console [ttyAMA2] enabled
```

- Here, ttyAMA0 and ttyAMA1 are the PL011 UART peripherals that are connected to the I/O virtualization block.
- Now test the PL011 UART peripherals by writing to the corresponding tty device files using echo command:

```
# echo "test message 0" > /dev/ttyAMA0
# echo "test message 1" > /dev/ttyAMA1
```

- The above commands print the message on the **FVP iomacro_terminal_0** and **FVP iomacro_terminal_1** terminals respectively.

PL330 DMA

There are two PL330 DMA controllers connected to the non-discoverable IO Virtualization block. Each of these controllers support 8 data channels and one instruction channel.

To test these dma controllers, DMA test guide included in the Linux kernel documentation has to be followed. As mentioned in the guide, CONFIG_DMATEST has to be enabled in the Linux kernel.

- After booting into busybox validate that the DMA PL330 controllers are probed fine and showing 8 channels on each dma controllers - dma0 and dma1.

```
# ls /sys/class/dma
dma0chan0  dma0chan3  dma0chan6  dma1chan1  dma1chan4  dma1chan7
dma0chan1  dma0chan4  dma0chan7  dma1chan2  dma1chan5
dma0chan2  dma0chan5  dma1chan0  dma1chan3  dma1chan6
```

- Also verify that the two dma controllers are attached to SMMUv3 of I/O Virtualization block. An example of this shown below.

```
# ls /sys/class/iommu/smmu3.0x0000000048000000/devices
ARMH0330:00  ARMH0330:01
```

- Following the DMA test guide, set the timeout and number of iterations. For example,

```
# echo 2000 > /sys/module/dmatest/parameters/timeout
# echo 1 > /sys/module/dmatest/parameters/iterations
```

- Start the test for different channels. For example, to run on dma0chan0 use the following command:

```
# echo dma0chan0 > /sys/module/dmatest/parameters/channel
# echo 1 > /sys/module/dmatest/parameters/run
```

- One can use loops to run the tests on all channels, for example for dma0:

```
# for ch in 0 1 2 3 4 5 6 7; do echo dma0chan$ch > /sys/module/dmatest/parameters/
↪channel; echo 1 > /sys/module/dmatest/parameters/run; sleep 1; done
```

  - Similarly, for other controller - dma1:

```
# for ch in 0 1 2 3 4 5 6 7; do echo dma1chan$ch > /sys/module/dmatest/parameters/
↪channel; echo 1 > /sys/module/dmatest/parameters/run; sleep 1; done
```

- Test result: Test results are printed to the kernel log buffer with the format:

```
"dmatest: result <channel>: <test id>: '<error msg>' with src_off=<val> dst_off=
↪<val> len=<val> (<err code>)"
```

  - Below example if from running the test on all channels of dma0.

```
# for ch in 0 1 2 3 4 5 6 7; do echo dma0chan$ch > /sys/module/dmatest/
↪parameters/channel; echo 1 > /sys/module/dmatest/parameters/run; sleep 1;␣
↪done
[ 1375.118694] dmatest: Added 1 threads using dma0chan0
[ 1375.118694] dmatest: Started 1 threads using dma0chan0
[ 1375.123195] dmatest: dma0chan0-copy0: summary 1 tests, 0 failures 235.45␣
↪iops 3060 KB/s (0)
```

(continues on next page)

(continued from previous page)

```
[ 1376.119025] dmatest: Added 1 threads using dma0chan1
[ 1376.119025] dmatest: Started 1 threads using dma0chan1
[ 1376.120894] dmatest: dma0chan1-copy0: summary 1 tests, 0 failures 615.76␣
↪iops 3078 KB/s (0)
[ 1377.119311] dmatest: Added 1 threads using dma0chan2
[ 1377.119311] dmatest: Started 1 threads using dma0chan2
[ 1377.123594] dmatest: dma0chan2-copy0: summary 1 tests, 0 failures 246.24␣
↪iops 2954 KB/s (0)

... and so on
```

SRAM Memory

There are six SRAM memory regions connected to the non-discoverable IO Virtualization block. Out of the six, two SRAM memory are connected to the the high bandwidth port of the I/O virtualization block and the remaining 4 are connected to the low bandwidth port. The size of each SRAM memory connected to the I/O virtualization block is 4MiB. The memory mapping for all the SRAM memory are listed in the below table:

SRAM that are connected to high bandwidth port:

Mem Name	Start Address	End Address	Size
MEM0	0x10_8001_0000	0x10_8001_FFFF	4MiB
MEM1	0x10_B002_0000	0x10_B002_FFFF	4MiB

SRAM that are connected to low bandwidth port:

```
iomacro_size = 0x2000000

iomacro_instance :
    RD-N2-Cfg1 = 1
    RD-N2      = 4
```

Mem Name	Start Address	End Address	Size
MEM2	0x4100_0000 + (*iomacro_instance* * *iomacro_size*)	0x413F_FFFF + (*iomacro_instance* * *iomacro_size*)	4MiB
MEM3	0x4140_0000 + (*iomacro_instance* * *iomacro_size*)	0x417F_FFFF + (*iomacro_instance* * *iomacro_size*)	4MiB
MEM4	0x4180_0000 + (*iomacro_instance* * *iomacro_size*)	0x41BF_FFFF + (*iomacro_instance* * *iomacro_size*)	4MiB
MEM5	0x41C0_0000 + (*iomacro_instance* * *iomacro_size*)	0x41FF_FFFF + (*iomacro_instance* * *iomacro_size*)	4MiB

- The SRAM memory can be tested by using **devmem** busybox utility which can be used to read and write to physical memory using **/dev/mem** provided that the Linux Kernel is built with Kernel config option **CONFIG_DEVMEM=y**
- After booting into busybox, use the following example to test the SRAM memory connected to the non-discoverable I/O virtualization block instance.
- Type "devmem" to display the busybox devmem utility info

```
# devmem
BusyBox v1.33.0 (2021-08-10 13:24:14 IST) multi-call binary.

Usage: devmem ADDRESS [WIDTH [VALUE]]

Read/write from physical address

        ADDRESS Address to act upon
        WIDTH   Width (8/16/...)
        VALUE   Data to be written
```

- Test SRAM memory write by using the following example.

```
# devmem 0x10800010000 32 0xabcd1234
```

- Similarly, test SRAM memory read by using the following example.

```
# devmem 0x10800010000 32
0xABCD1234
```

This completes the testing for non-PCI devices connected to the I/O virtualization block.

33.4 PCIe I/O virtualization

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.4.1 What is I/O virtualization?

I/O virtualization allows sharing a common I/O resource between multiple running virtual machines so that the resource usage and cost are optimized for a typical infrastructure use-case. Few techniques used for I/O virtualization are:

- Trap and emulate
- Paravirtualization
- PCI passthrough

This page describes the PCI passthrough technique that is the most widely adopted technique for I/O virtualization.

33.4.2 PCIe pass-through based device virtualization

PCIe pass-through (also called as direct device assignment) allows a device to be assigned to a guest such that the guest runs the driver for the device without intervention of the hypervisor/host. This is one of the device virtualization technique besides para-virtualization.

PCIe pass-through is achieved using frameworks in Linux kernel, such as VFIO, virtio, IOMMU, and pci. A smmu-test-engine (smmute) device that is available on the platform is used as a test device for this virtualization technique. The smmu-test-engine is a PCIe exerciser that generates DMA workloads and it uses arm-smmu-v3 to provide dma isolation. This device first probed in the host kernel can be assigned to the guest and the smmu-test-engine driver in the guest kernel can then manage the device directly.

PCI pass-through using multiple guests and smmu test engine:

- Boot the platform by following the *Buildroot* guide, and then ensure that the smmu test engine device is probed correctly. Use the lspci command to check for smmu test engine devices with pci BDF ids - 07:00.0, 07:00.3, 08:00.0 and 08:00.1.

```
lspci
```

- Verbose output of lspci will show the last four devices with above mentioned pci BDF ids are managed by 'smmut-pci' kernel driver.

```
lspci -v
```

- Also check that the smmute-pci driver has probed the smmu test engine devices properly, and a device entry exists for each of the four smmute devices.

```
ls -l /dev/smmute*
```

- Use one of the smmute devices (e.g. device 0000:08:00.1) to perform the PCI pass-through. Detach the pcie device from its class driver and attach to vfio-pci driver, as also explained in the kernel doc.

```
echo 0000:08:00.1 > /sys/bus/pci/devices/0000:08:00.1/driver/unbind
echo vfio-pci > /sys/bus/pci/devices/0000:08:00.1/driver_override
echo 0000:08:00.1 > /sys/bus/pci/drivers_probe
```

- The kernel and ramdisk images to launch VMs are available in the second partition of grub disk image that gets probed at /dev/vda2 in the host. Mount this to use the images.

```
mount /dev/vda2 /mnt
```

- This mounted partition can also be shared with guest using 9p virtual filesystem. A binary to run tests over smmute device is also available in this partition. So after sharing the filesystem with a guest, tests can be run on assigned smmute device to verify pci pass-through.

- Launch VMs using lkvm tool that supports virtio-iommu and vfio drivers to allow pci pass-through.

```
screen -md -S "virt1" /mnt/kvmtool/lkvm run -k /mnt/Image -i /mnt/ramdisk-
↪buildroot.img --irqchip gicv3-its -c 2 -m 512 --9p /mnt,hostshare --
↪console serial --params "console=ttyS0 --earlycon=uart,mmio,0x1000000␣
↪root=/dev/vda" --vfio-pci 0000:08:00.1 --disable-mte;
```

- Jump to the right screen to view boot-up logs from guest. Use following command to go to a specific screen:

```
screen -r virt1
```

- After the guest boots up, mount the 9p filesytem to a mount point in the guest. For example, use the following command to mount at /tmp

```
mount -t 9p -o trans=virtio hostshare /tmp/
cd /tmp
```

- Check that the smmu test engine is probed in the guest. The device will show a different pci BDF id here in guest as compared to the id shown in host kernel.

```
# lspci
00:00.0 Unassigned class [ff00]: ARM Device ff80
```

(continues on next page)

(continued from previous page)

```
# ls -l /dev/smmute*
crw-------    1 root     root      235,   0 Jan  1 00:00 /dev/smmute0
```

- From /tmp directory that contains the 'smmute' binary, run the test.

```
./smmute -s 0x100 -n 10
```

- Check that the MSI interrupts on the smmu test engine PCI device in the guest are triggered.

```
cat /proc/interrupts
```

  - For example, after running few iterations of smmute test the MSI interrupts on the PCI device would look like:

```
#          CPU0       CPU1       CPU2       CPU3
20:          1          0          0          0   ITS-MSI   0 Edge      ␣
↪0000:00:00.0
21:          0          2          0          0   ITS-MSI   1 Edge      ␣
↪0000:00:00.0
22:          0          0          1          0   ITS-MSI   2 Edge      ␣
↪0000:00:00.0
23:          0          0          0          1   ITS-MSI   3 Edge      ␣
↪0000:00:00.0
24:          1          0          0          0   ITS-MSI   4 Edge      ␣
↪0000:00:00.0
25:          0          1          0          0   ITS-MSI   5 Edge      ␣
↪0000:00:00.0
26:          0          0          1          0   ITS-MSI   6 Edge      ␣
↪0000:00:00.0
27:          0          0          0          0   ITS-MSI   7 Edge      ␣
↪0000:00:00.0
```

- Jump back to the host by exiting the screen using 'Ctrl-a d', and launch another guest by repeating the above commands and updating the screen_name, and device. For example,

```
echo 0000:08:00.0 > /sys/bus/pci/devices/0000:08:00.0/driver/unbind
echo vfio-pci > /sys/bus/pci/devices/0000:08:00.0/driver_override
echo 0000:08:00.0 > /sys/bus/pci/drivers_probe

screen -md -S "virt2" /mnt/kvmtool/lkvm run -k /mnt/Image -i /mnt/ramdisk-
↪buildroot.img --irqchip gicv3-its -c 2 -m 512 --9p /mnt,hostshare --
↪console serial --params "console=ttyS0 --earlycon=uart,mmio,0x1000000␣
↪root=/dev/vda" --vfio-pci 0000:08:00.0 --disable-mte;
```

- Perform test over smmu test engine in this second screen by mounting the 9p filesystem and executing the 'smmute' binary. Check again in this guest that the MSI interrupts on the smmu test engine PCI device are triggered.

```
cat /proc/interrupts
```

- Jump back to the host by exiting the screen using 'Ctrl-a d' and use the following command to list the guests that are managed by lkvm tool.

```
# /mnt/kvmtool/lkvm list
PID NAME                 STATE
------------------------------------
309 guest-309            running
276 guest-276            running
```

- Power-off the guests by jumping to the respective screens and executing the command:

```
poweroff
```

- The guests would shutdown and the following message would be displayed on the console.

```
# KVM session ended normally.
```

33.5 Virtual Interrupts And VGIC

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.5.1 Overview of Directly Injected vLPIs

Locality-specific Peripheral Interrupts (LPIs) are message based interrupts which are raised on particular targeted processing elements (PEs) only. These interrupts do not use any physical lines, hence they need additional hardware (H/W) support for raising an event. Arm Generic Interrupt Controller (GIC) Interrupt Translation Service (GIC-ITS) hardware provides such support by accepting a MMIO write and raising an interrupt on the target PE. With the advancement in GIC-ITS and rising need of LPIs in virtualization, the support for directly injected virtual LPIs (vLPIs) was added in GICv4. With GICv3 and GICv3-ITS (GIC version 3 with support for ITS hardware) the virtual interrupts injection into the guest VM is done by writing into the GIC List Registers (LRs) which are part of virtualized GIC cpu interface. But use of LRs to inject virtual interrupts calls for hypervisor intervention every time a physical interrupt is triggered. With KVM hypervisor the LRs are updated only at the next scheduled run of the guest on any physical PE. This introduces further delay in interrupt handling in a guest environment.

In GICv4 ITS a new set of redistributor registers are added to hold the addresses of LPI configuration and LPI pending tables of the running VM. These registers are banked for each redistributor corresponding to each PE. Similarly, a new ITS table called as virtual PE (vPE) table is added. This table is equivalent to collection tables used for physical LPIs.

A new set of ITS commands is also added to update the ITS device table, interrupt translation table and the vPE table along with redistributor's configuration and pending tables. With these additions the KVM hypervisor now has to configure these ITS tables only once at the beginning and thereafter whenever a message based physical LPI is raised by a peripheral, GIC-ITS H/W looks up the tables to find any corresponding virtual LPI entry and updates it to the redistributor of the target vPE. From there on redistributor is responsible to trigger it to PE. This avoid any requirement of software (KVM) intrusion and makes it almost immediate trigger of vLPIs.

33.5.2 Overview of Directly Injected vSGIs

Software Generated Interrupts (SGIs) are typically used for inter-processor communication among the PEs. As the name suggests, it is generated by software by writing to the GIC cpu interface registers. Software running on one PE writes to one of the per PE banked vsgi register of GIC cpu interface. During the write it provides information about the interrupt ID and the target PE the interrupt is meant for. With older gicv3 and gicv3-its only way for KVM to handle this is to trap the write to SGI register from sender and updates list registers (LRs) to inject it into guest VM which is deferred until the VM rescheduled on the target PE. This problem of deferred interrupts was solved with support of direct vSGI injection using GIC-ITS H/W as offered in GICv4.1. A new GIC-ITS command was added to hold entries of vSGIs configurations for sending vPE. Also a new GIC-ITS register was introduced which can be used to raise vSGI by simply writing to it. And extra redistributor registers to poll the state of vSGIs on target vPEs was also added. With direct vSGI injection, now whenever sender PE writes to SGI register of GIC cpu interface to raise interrupt to target PE, it is trapped by KVM and then a write to one of the GIC-ITS register is done, which immediately raises the interrupt to target vPE, skipping the need to wait until rescheduling of the guest VM and thus avoiding any delays.

33.5.3 Build & Install

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Build the platform software

This section describes the procedure to build the software stack required to perform KVM unit testing. Following software packages from the Neoverse reference platform software stack are needed to do the testing:

- Software stack for distro boot as given in *Distro Boot* guide,
- Refinfra Linux and smmu-test-engine tools.
- kvm-unit-tests built for kvmtool target,
- Kvmtool VMM.

All the above package can be compiled together by buildroot build. Proceed by running the appropriate script from software stack

```
./build-scripts/rdinfra/build-test-buildroot.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`
    - `build`
    - `package`
    - `all` (all of the three above)

Examples of the build command are

- Command to clean, build and package the software stack for the RD-N2-Cfg1 platform:

```
./build-scripts/rdinfra/build-test-buildroot.sh -p rdn2cfg1 all
```

Setup Satadisk Images

The direct injection of vLPI and vSGI can be validated on a Linux distributions running as the host OS. Create disk images by following the guidelines from *Distro Boot* page.

Note: For simplicity, the setup instructions where specific, are given for Ubuntu distro host OS.

- Boot the host satadisk image on the FVP with network enabled as mentioned in *Distro Boot*. For example, to boot Ubuntu as the host OS give the follwing command to begin the distro boot from the `ubuntu.satadisk` image:

```
./distro.sh -p rdn2cfg1 -d /absolute/path/to/ubuntu.satadisk -n true
```

- Once the host OS is booted up ensure that the KVM and virtualization support is enabled. After booting enable the networking support as well. Follow the *UEFI supported virtualization guide* for details on preparing the setup with Linux distribution running as host OS with networking enabled. For example, one might need to run the following commands:

```
sudo dhclient -v
sudo apt update
sudo apt install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils net-
↪tools libfdt-dev -y
```

Note: Below step can be skipped if the host Ubuntu distro version is v22.04 or above because it uses linux version 5.15.0 which already has support for GICv4.

- For the direct injection vSGI test, GICv4 driver support is required in linux kernel. This is achieved by installing the `refinfra` linux kernel to the host OS distribution which is temporarily realized by copying the kernel to the host `/boot/` directory as shown below.

```
sudo rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/
↪linux/Image /boot/vmlinuz-refinfra
```

Note: This is not a recommended way to install a new kernel to Ubuntu. This approach is chosen only for quick kvm testing and doesn't guarantee stable Ubuntu after the installation.

- Under default kernel setup direct injection of vLPI and vSGI isn't activated in KVM. And this is activated by enabling kernel boot parameter *kvm-arm.vgic_v4_enable*. Also to enable display of grub menu during boot make the necessary changes to specific variables in the user grub config file */etc/default/grub* as shown below.

```
#Before Change->
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=0
GRUB_CMDLINE_LINUX_DEFAULT="..."
#GRUB_TERMINAL=console
```

(continues on next page)

(continued from previous page)

```
#After Change->
GRUB_TIMEOUT_STYLE=menu
GRUB_TIMEOUT=10
GRUB_CMDLINE_LINUX_DEFAULT="... kvm-arm.vgic_v4_enable=1"
GRUB_TERMINAL=console
```

- To reflect all the changes related to grub config and create grub menuentry for the new *refinfra* ` kernel. Do a grub update and shutdown the host.

```
sudo update-grub
sudo poweroff
```

33.5.4 Running The Test

vSGI Test

- It is necessary to choose right version of kernel while booting the host satadisk image for this test from the GRUB boot menu at the boot time. So go ahead and boot the host satadisk image on the FVP as mentioned in *Distro Boot*. For host Ubuntu distro version below v22.04, ensure to select menuentry **"Ubuntu, with Linux refinfra"** from sub-menuentry **"Advanced options for Ubuntu"**. Command to begin the Ubuntu distro boot from the `ubuntu.satadisk` image:

```
./distro.sh -p rdn2cfg1 -d /absolute/path/to/ubuntu.satadisk -n true
```

- Executing the testcase will require the `kvm-unit-tests` directory, and the `kvmtool` binary which were built in section *Build the platform software*. Copy these to host OS through network and run the test

```
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/kvm-ut .
cd kvm-ut/
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/rdn2/
↪lkvm .

sudo ./lkvm run -m 2048 -f arm/gic.flat --irqchip gicv3-its -p "ipi"
```

If all the tests passes, the logs should output concluding successful completion of vSGI testing.

```
PASS: gicv3: ipi: self: Interrupts received
PASS: gicv3: ipi: target-list: Interrupts received
PASS: gicv3: ipi: broadcast: Interrupts received
SUMMARY: 3 tests
```

- Shutdown the running host OS and move on to the next test.

```
sudo poweroff
```

vLPI Test

- It is necessary to choose right version of kernel while booting the host satadisk image for this test from the GRUB boot menu at the boot time. It is essential to avoid booting with `refinfra` kernel and rather use any other kernel version. So go ahead and boot the host satadisk image on the FVP as mentioned in *Distro Boot*. For host Ubuntu distro version below v22.04, ensure to select any menuentry other than **"Ubuntu, with Linux refinfra"** from sub-menuentry **"Advanced options for Ubuntu"**. Command to begin the Ubuntu distro boot from the `ubuntu.satadisk` image:

```
./distro.sh -p rdn2cfg1 -d /absolute/path/to/ubuntu.satadisk -n true
```

- Neoverse reference platforms have few smmu-test-engine devices that are the PCIe endpoint devices that can be used to demonstrate this feature. For this test, one of the smmu-test-engine (smmute) from I/O macro block is used to generate vLPIs. And the generated vLPI is received by a guest virtual machine (VM) running the `refinfra` linux kernel with support of smmute driver. To setup a guest virtual machine, KVM hypervisor is employed here. To learn more in detail about KVM and virtualization read through *Virtualization using KVM* and *UEFI supported virtualization guide*.

  Running the KVM session will require the `refinfra Linux kernel` image, the `ramdisk-buildroot.img` initrd image and the `kvmtool` binary. vLPI test will require the smmute testapp `smmute` be executed from guest. Create a test workplace and download all the built binaries and images.

```
mkdir -p ~/vlpi-test;
cd ~/vlpi-test
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/ramdisk-buildroot.
↪img .
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/linux/
↪Image .
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/linux/
↪tools/iommu/smmute/smmute .
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/rdn2/
↪lkvm .
```

- Run the below command to attach the smmute device to `vfio-pci` driver on host. This is required to allow PCI endpoint device passthrough to the guest OS. Please follow through the below commands to quickly setup the device and to learn more in detail about it, read through Linux vfio.

```
sudo modprobe vfio-pci
echo "vfio-pci" | sudo tee /sys/bus/pci/devices/0000\:08\:00.1/driver_override
echo "0000:08:00.1" | sudo tee /sys/bus/pci/drivers_probe
```

- Make sure that the device is attached to vfio-pci driver.

```
$ lspci -vv -s 0000:08:00.1 |grep vfio-pci
     Kernel driver in use: vfio-pci
```

- Launch the virtual machine with a kernel image and initrd image as the guest OS.Run the below command from `vlpi-test` workspace directory to start a KVM session with kernel image `Image`, initrd image `ramdisk-buildroot.img` and the PCI device with requester-ID (BDF) `0000:08:00.1` used for direct device assignment:

```
screen -md -S "virt0" sudo ./lkvm run -m 2048 -k Image -i ramdisk-buildroot.img --
↪irqchip gicv3-its --9p $(pwd),hostshare --console serial -p "console=ttyS0 --
↪earlycon=uart,mmio,0x1000000 ip=dhcp" --vfio-pci 0000:08:00.1 --disable-mte;␣
↪screen -r virt0;
```

- Enter sudo password if prompted for one.
- After the guest boots up, mount the 9p filesytem with mount_tag `hostshare` to discover the `smmute` testapp in the guest and finally run the smmute testapp as shown below:

```
mount -t 9p -o trans=virtio hostshare /tmp/
cd /tmp

./smmute -s 0x100 -n 10
```

Running the test, outputs the log similar to what is shown below for 10 transactions. If all the transactions has status 0 (success) without any popping kernel log about missed MSI-X transaction, it is safe to say direct injection of vLPI is tested.

```
Result:
- transaction          = 2
- status               = 0 Success
- value                = 0x0
- duration             = 2 us
Output buffer:
000: f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff 00
010: 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10
020: 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
030: 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30
...
```

- At last shutdown the guest

```
poweroff
```

And on completion of guest shutdown `kvmtool` prints a message denoting error free closing of KVM session.

```
# KVM session ended normally.
```

33.6 UEFI Based KVM Virtualization

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

33.6.1 Overview of Virtualization support

Neoverse reference platforms support virtualization by providing architectural support of AArch64 virtualization host extension (VHE). The reference platform software stack uses Linux kernel based virtual machine (KVM) as the hypervisor and the userspace program kvmtool as the virtual machine manager (VMM) to leverage this hardware feature. The *Virtualization document* guides on how to validate virtualization on Neoverse reference platforms using a buildroot filesystem with Linux as the guest operating system. This setup helps in validating the architectural features, however lacks the support of a firmware to boot the platform. Booting a full fledged Linux distribution operating system (OS) such as Fedora or Ubuntu, etc. with UEFI firmware and grub boot-loader as the guest OS can help in validating more real-time virtualization use-cases. This setup also provides support for ACPI tables based platform resource control.

33.6.2 Objective

The purpose of validating virtualization with a Linux distribution is to prepare virtual machines (VM) on a host system that allow booting multiple guest operating systems running Linux distributions such as Ubuntu, Fedora, etc. with the UEFI firmware support. The virtualized platform is prepared and launched using KVM module of the host Linux kernel and *kvmtool* which is a standalone userspace tool. *kvmtool* allows booting either directly from a kernel or from a firmware, where firmware will initiate the bootloader for Linux distro OS boot. The firmware based booting allows inclusion of ACPI tables to communicate the hardware info to the OS and perform resource control. The firmware is built with the UEFI EDK2 *ArmVirtKvmTool* platform descriptor from *ArmVirtPkg* EDK2 package. The ArmVirtKvmTool takes help of *DynamicTablesPkg* EDK2 package to dynamically produce ACPI tables from device tree blob (dtb). The *DynamicTablesPkg* parses the harware information from the dtb that is prepared by the kvmtool for the spawned VMs.

The spawned virtual machine simulates the necessary hardware required for the guest to run. This hardware support includes, but not limited to:

- Processor (vCPUs)
- Interrupt controller (e.g. gic-v3, gic-v3-its)
- Main memory or RAM
- Timer (e.g. armv8/7-timer)
- Flash memory (e.g. cfi-flash) required by UEFI firmware
- UART controller (e.g. uart-16550) to setup console devices,
- Real time clock (e.g. motorola,mc146818)
- Block and net devices for disk access and network support both of which are realised using virtio devices.

It is important to note that for this validation all the virtio devices (block and net devices) use pci as their underlying transport mechanism and thus are enumerated as pci endpoint devices.

33.6.3 Overview of ArmVirtKvmTool

ArmVirtKvmTool firmware is specifically designed to initialize the hardware (h/w) that is described by the kvmtool using device tree during the VM launch. The ArmVirtKvmTool supports multiple libraries corresponding to the hardware devices emulated by kvmtool, e.g. flash memory, uart, rtc, timer, pci and virtio devices. Few common devices that require initialization by the firmware are parsed through flattened device tree (fdt) library. The firmware also makes use of *KvmtoolVirtMemInfoLib* library to create a system memory map before doing the h/w initization. The ArmVirtKvmTool platform descriptor is originally based on *ArmVirtPkg* and borrows various base libraries to implement the pre-pi and dxe stage drivers.

EDK2 supports handling ACPI tables which are then passed to OS after firmware exits from bds stage. But as kvmtool provide h/w info as dtb and not as ACPI tables, another EDK2 package *DynamicTablePkg* is used to dynamically parse the dtb and generate appropriate ACPI tables. *ArmVirtKvmTool* implements a configuration manager protocol that holds a platform info repository. The fdt hardware parser from *DynamicTablePkg* is used to collect all the platform details as Arm Cmobjects and then to communicate these objects to the table factory of *DynamicTablePkg*. The table factory obtains a rich set of ACPI table generators from the main table manager and sequentially invokes each generator to create a table. The supported list of libraries include DBG2, FADT, GTDT, IORT, MADT, MCFG, PPTT, SPCR and many more.

It is equally important to align the firmware input based on the environment created by *kvmtool* with the help of KVM. Refer the *Virtualization document* for more details on configuring kvmtool for the required virtual platform.

33.6.4 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

This section describes the procedure to prepare the necessary setup to validate UEFI firmware based booting of Linux distributions on the virtual machines. Following software packages from the Neoverse reference platform software stack are needed to do the validation:

- ArmVirtKvmTool based firmware (built as part of UEFI build)
- Kvmtool VMM

Skip this section if a *Buildroot* or *Busybox* build is already performed for the platform software stack as the `ArmVirtKvmTool` uefi firmware and `kvmtool` binaries are already built.

- Build UEFI firmware for the host and for the guest OS (`ArmVirtKvmTool`) by running the appropriate script from software stack:

```
./build-scripts/build-test-uefi.sh -p <platform name> <command>
```

Supported command line options are listed below

- <platform name>
  - Lookup for a platform name in Platform Names.
- <command>
  - Supported commands are
    - `clean`
    - `build`
    - `package`
    - `all` (all of the three above)

Examples of the build command are

- Command to clean, build and package the software stack needed for the UEFI firmware on RD-N2-Cfg1 platform:

```
./build-scripts/build-test-uefi.sh -p rdn2cfg1 all
```

- Lastly, build the userspace hypervisor program `kvmtool`.

```
./build-scripts/build-kvmtool.sh -p <platform name> clean
./build-scripts/build-kvmtool.sh -p <platform name> build
./build-scripts/build-kvmtool.sh -p <platform name> package
```

- <platform name>
  - Lookup for a platform name in Platform Names.

For examples to build kvmtool for rdn2cfg1 platform use the below command:

```
./build-scripts/build-kvmtool.sh -p rdn2cfg1 clean
./build-scripts/build-kvmtool.sh -p rdn2cfg1 build
./build-scripts/build-kvmtool.sh -p rdn2cfg1 package
```

33.6.5 Setup Satadisk Images

To use Linux distributions as the host and guest OS create disk images by following the guidelines from *Distro Boot* document. There can be a Ubuntu or Fedora as host OS and multiple distributions as guest. It is important to remember however, that the host disk image should be large enough to hold multiple guest disk images e.g. host of ~32GiB and multiple guest images of Ubuntu/Fedora with ~6GiB size. Guest disk images are used later to run KVM session.

Note: For simplicity the setup instructions where specific are given for Ubuntu v22.04 distro host OS.

33.6.6 Booting the platform for validation

Boot Host OS

- Boot the host satadisk image on the FVP with network enabled as mentioned in *Distro Boot*. For example, to boot Ubuntu as the host OS give the follwing command to begin the distro boot from the `ubuntu.satadisk` image:

```
./distro.sh -p rdn2cfg1 -d /absolute/path/to/ubuntu.satadisk -n true
```

- After booting the host OS verify that the KVM and virtualization support is enabled. Each Linux distro has different ways to verify this but it is also possible to confirm by looking into the kernel boot logs.

```
dmesg | grep -i "kvm"
```

  Above command puts out KVM related boot logs which should be similar to the logs shown below:

```
kvm [1]: IPA Size Limit: 48 bits
kvm [1]: GICv4 support disabled
kvm [1]: GICv3: no GICV resource entry
kvm [1]: disabling GICv2 emulation
kvm [1]: GIC system register CPU interface enabled
kvm [1]: vgic interrupt IRQ1
kvm [1]: VHE mode initialized successfully
```

  Also make sure `/dev/kvm` exists. If any of this is not met, please follow through for the solution mentioned in the below sections.

Network Support

- Check if host OS has network access by running `ping -c 5 8.8.8.8`. If the ping doesn't work as the network is unreachable then enable it using `dhclient` utility for dhcp discovery on the host OS:

```
sudo dhclient -v
```

- Check the available network interfaces on the host with below command:

```
ip link show
```

  Check if the above command shows a virtual bridge `virbr#` already configured and running on host. This virtual bridge will help in giving network access to the guest OS.

- If the KVM support or the virtual bridge could not be found then try the below commands. For more details refer to the instructions in Ubuntu KVM Installation guide to resolve any issues.

```
sudo apt update
sudo apt install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils libfdt-
↪dev -y
```

- Now start the `libvirtd` service to initiate the communication between the KVM and the libvirt APIs. Use below commands to configure the system to start the service at every boot.

```
sudo systemctl start libvirtd
sudo systemctl enable libvirtd
```

- The network acces to the guest OS can be given by creating a bridge and a tap interface. Follow commands shown below to create the tap interface and add it to virtual bridge `virbr#` as listed from executing `ip link show`.

```
sudo ip tuntap add dev tap0 mode tap user $(whoami)
sudo ip link set tap0 master virbr# up
```

Now create a workspace to begin with virtualization test example.

```
mkdir -p ~/kvm-test/
cd ~/kvm-test/
```

Emulate Flash Memory

`ArmvirtKvmTool` UEFI firmware needs a flash memory while booting to store various objects. Create an empty zero filled flash memory file which will be presented by kvmtool as a flash device to the UEFI firmware and guest OS.

```
dd if=/dev/zero of=efivar.img bs=128M count=1
```

Enable PCIe pass-through based device virtualization

As mentioned in the *Virtualization document* PCIe pass-through (also called as direct device assignment) allows a device to be assigned to a guest such that the guest runs the driver for the device without intervention of the hypervisor/host. This is one of the device virtuali- zation technique that provides near near host device performance. This is achieved with the help of VFIO driver framework and IOMMU support. More about this can be read from Linux vfio.

- Neoverse reference platforms have few smmu-test-engine devices that are the PCIe endpoint devices that can be used to demonstrate this feature Use the verbose `lspci` command to check the status of these devices for example, with pci BDF ids 08:00.0 and 08:00.1.

```
sudo lspci -v
sudo lspci -v -s 0000:08:00.1
```

- Check if `vfio_pci` kernel module is already loaded or not.

```
lsmod | grep -i "vfio"
```

  if not then manually probe the kernel driver module

```
sudo modprobe vfio-pci
```

- Unbind the pci endpoint device from its current driver if the device is attached to its class driver. If the driver doesn't exist ignore the error produced on running below command

```
echo "0000:08:00.1" | sudo tee /sys/bus/pci/devices/0000\:08\:00.1/driver/unbind
```

- Bind the device to `vfio-pci` driver

```
echo "vfio-pci" | sudo tee /sys/bus/pci/devices/0000\:08\:00.1/driver_override
echo "0000:08:00.1" | sudo tee /sys/bus/pci/drivers_probe
```

- Confirm that device has been attached to `vfio-pci` driver

```
sudo lspci -v -s 0000:08:00.1 | grep -i "Kernel driver"
```

- In order to use the device for direct assignment, it is required that all the devices sharing the iommu group with this particular device are attached to `vfio-pci` driver. So perform the above mentioned unbinding and binding for all the endpoint devices that shares the common iommu group. List out all the devices that are under that specific iommu group

```
ls /sys/bus/pci/drivers/vfio-pci/0000\:08\:00.1/iommu_group/devices/
```

Obtain the built binaries

- Running the KVM session will require the `ArmvirtKvmTool` UEFI firmware, a guest disk image with pre-installed Linux distro OS and the `kvmtool` binary which were obtained in section *Build the platform software*. Copy these to the host OS through network using below commands in the workspace directory `kvm-test`.

```
rsync -Wa --progress user@server:absolute/path/to/guest-ubuntu.satadisk .
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/css-
↪common/KVMTOOL_EFI.bin .
rsync -Wa --progress user@server:TOP_DIR/output/<platform name>/components/kvmtool/
↪lkvm .
```

Launch VMs with multiple Linux distributions

Finally, launch the virtual machine with a Linux distribution image as the guest OS. As mentioned in the *Virtualization document* 'screen' utility can be used to multiplex console outputs.

Note: To switch back to host session detach from the screen by pressing `ctrl+a d`.

Run the below command from `kvm-test` workspace directory to start a KVM session with ArmvirtKvmTool binary `KVMTOOL_EFI.bin`, kvmtool binary `lkvm`, flash image `efivar.img`, the distribution disk image for guest `guest-ubuntu.satadisk`, `tap0` tap inteface and the PCI device with requester-ID (BDF) `0000:08:00.1` used for direct device assignment:

```
screen -md -S "virt0" sudo ./lkvm run -m 2048 -f KVMTOOL_EFI.bin -F efivar.img -d guest-
↪ubuntu.satadisk -n tapif=tap0 --console serial --force-pci --vfio-pci 0000:08:00.1 --
↪disable-mte;
```

- The launched screens can be viewed from the target by using the following command:

```
screen -ls
```

- Jump to the screen using:

```
screen -r virt0
```

- The guest can be seen booting with logs as shown below:

```
# lkvm run --firmware ./KVMTOOL_EFI.bin -m 2048 -c 4 --name guest-3882
Info: Using IOMMU type 3 for VFIO container
Info: 0000:08:00.1: assigned to device number 0x0 in group 3
Info: flash file size (134217728 bytes) is not a power of two
Info: only using first 16777216 bytes
UEFI firmware (version  built at 14:51:31 on Apr  4 2022)
```

- Notice the logs about PCIe device being setup using the Linux VFIO driver.

```
Info: Using IOMMU type 3 for VFIO container
Info: 0000:08:00.1: assigned to device number 0x0 in group 9
```

- Once the guest has booted. check if network is accessible and assigned pci device is listed in `lspci`.

```
# If network is unreachable use dhclient:
sudo dhclient -v

ping -c 2 8.8.8.8

# Check the listed PCI devices
lspci

# Output of lspci
00:00.0 Unassigned class [ff00]: ARM Device ff80
```

- To shutdown the guest execute the following command:

```
sudo poweroff
```

  On completion of guest shutdown `kvmtool` prints a message denoting error free closing of KVM session.

```
# KVM session ended normally.
```

CHAPTER

THIRTYFOUR

VIRTIO-P9

Important: This feature might not be applicable to all Platforms. Please check individual Platform pages, section **Supported Features** to confirm if this feature is listed as supported.

34.1 Overview of P9 filesystem

9P (or the Plan 9 Filesystem Protocol) is a network protocol developed for the Plan 9 from Bell Labs distributed operating system as the means of connecting the components of a Plan 9 system. As mentioned at lwn 9P is somewhat equivalent to NFS or CIFS, but with its own particular approach. It is not as much a way of sharing files as a protocol definition aimed at the sharing of resources in a networked environment. It works in a connection-oriented manner in which each client makes one or more connections to the server(s) of interest. The client can create file descriptors, use them to navigate around the filesystem, read and write files, create, rename and delete files, and close things down.

34.2 Overview of Virtio-P9 device

Few Arm reference design Fixed Virtual Platforms (FVPs) for infrastructure implement a subset of the Plan 9 file protocol over a virtio transport. This component is called Virtio-P9 device and it enables accessing a directory on the host's filesystem within Linux, or another operating system that implements the protocol, running on a platform model. Put simply 9P filesystem protocol enables communicating the file I/O operations between guest systems or clients and the 9p server.

Linux running on the host uses the v9fs which is a Unix implementation of the Plan 9 9p remote filesystem protocol, in conjunction with the virtio transport protocol to allow filesystem I/O operations between host and the FVP.

As mentioned on the Arm Fixed Virtual Platform page (FVP) the component implements a subset of the Linux 9P2000.L protocol with the limitation that the guest can mount only one host directory per instance of the component.

34.3 Build the platform software

Note: This section assumes the user has completed the chapter *Getting Started* and has a functional working environment.

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

34.4 Running the test to validate Virtio-P9 device

- To begin validating the Virtio-P9 device create a directory on the host Linux machine from which the target platform FVP is launched. This directory is used as a shared directory between the host and target FVP.

```
mkdir /tmp/hostsharedir
```

- Copy few files that can be shared to the target platform into this hostsharedir. The files can be read/written from the booted target platform for validating Virtio-P9.
- To enable Virtio-P9 device on the platform pass the following additional parameter when launching the target platform FVP:

```
-C board.virtio_p9.root_path=<Path_to_shared_dir>
```

Example,

```
-C board.virtio_p9.root_path=/tmp/hostsharedir
```

- As mentioned in the *Busybox Boot* guide boot to busybox using the commands mentioned below.

```
./boot.sh -p <platform name> -a <additional_params> -n [true|false]
```

Here the supported command line options are:

- -p <platform name>
  - Lookup for a platform name in Platform Names.
- -n [true|false] (optional)
  - Controls the use of network ports by the model. If network ports have to be enabled, use 'true' as the option. Default value is set to 'false'.
- -a <additional_params> (optional)
  - Specify any additional model parameters to be passed. The model parameters and the data to be passed to those parameters can be found in the FVP documentation.

Example command to boot a RD-N2-Cfg1 platform upto busybox prompt with Virtio-P9 device enabled:

```
./boot.sh -p rdn2cfg1 -a '-C board.virtio_p9.root_path=/tmp/hostsharedir'
```

- Once the platform is booted mount the 9P filesystem from busybox prompt:

```
mount -t 9p -o trans=virtio,version=9p2000.L FM <mount_point>
```

Example,

```
mount -t 9p -o trans=virtio,version=9p2000.L FM /mnt
```

- Now access the files present in the mounted /mnt directory and verify that the files can be read from and written to.
- Try creating a new test file in the mounted path to transfer some data from the booted target platform to the host PC.

```
dmesg > /mnt/kernel_logs.txt
```

- Try to access the shared directory on the host PC to verify that the file created on the target platform is also visible in host PC.

```
cat /tmp/hostsharedir/kernel_logs.txt
```

- Once the file accesses are validated between the host PC and target FVP platform unmont the 9P filesystem from the target platform's busybox prompt.

```
umount /mnt
```

This completes the validation of Virtio-P9 component on Arm infrastructure reference design platforms.

CHAPTER

THIRTYFIVE

RD-INFRA-2025.02.04

35.1 Release Description

This release fixes the Realm state virtual machine launch that was reported as not working in *RD-INFRA-2024.12.20*. The fixes are provided in the RMM component.

- *RD-V3-R1*
- *RD-V3-R1-Cfg1*
- *RD-V3-Cfg2*
- *RD-V3-Cfg1*
- *RD-V3*

FVP versions:

- RD-V3-R1 & RD-V3-R1-Cfg1 : 11.27.51
- RD-V3, RD-V3-Cfg1 & RD-V3-Cfg2 : 11.27.51

35.2 Change Log

TF-M:

- No updates

SCP:

- No updates

TF-A:

- No updates

RMM:

- Fix to hide MPAM from Realm state

Hafnium:

- No updates

edk2:

- No updates

edk2-platforms:

- No updates

Linux:

- Update to cca/v6

kvmtool

- Update to cca/v4

kvm-unit-tests:

- No updates

build-scripts:

- No updates

container-scripts:

- No updates

model-scripts:

- No updates

buildroot:

- No updates

35.3 Supported Features

- No updates

35.4 Known Limitations

- For RD-V3-Cfg2, boot times have increased and it is suggested to use HEADLESS mode as a workaround by using the -j option with the boot scripts. Example: *./boot.sh -p rdv3cfg2 -j*. This will not launch any UART xterm windows, but the UART logs will be captured in the log file.
- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8 GiB. Therefore, total DRAM Memory for RD-V3-Cfg2 is limited to 8 GiB to support Realm VMs and Realm KVM unit test.
- *LocateHandleBuffer_Func* tests of UEFI SCT test suite, which are executed as part of the SystemReady Compliance Program are experiencing prolonged execution times and the suite may timeout before test completion.

35.5 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V3-R1
  - Virtual machine boot in Realm state, KVM-UT
- RD-V3-R1-Cfg1
  - Virtual machine boot in Realm state, KVM-UT
- RD-V3-Cfg2
  - Virtual machine boot in Realm state, KVM-UT
- RD-V3-Cfg1
  - Virtual machine boot in Realm state, KVM-UT
- RD-V3
  - Virtual machine boot in Realm state, KVM-UT

35.6 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.12.20
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.12.20
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.12.20
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2025.02.04
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git
  - Tag/Hash : 41e8d5b1f805e882554b567e587c0eed5a81c49d
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.12.20

- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.12.20
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2025.02.04
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : G20240322
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-3.6.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_1
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2024.12.20
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Hash : cca/v4
- KVM unit tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests
  - Tag/Hash : RD-INFRA-2024.12.20

CHAPTER

THIRTYSIX

RD-INFRA-2024.12.20

36.1 Release Description

Software stack refreshed for the following platforms:

- *RD-V3-R1*
- *RD-V3-R1-Cfg1*
- *RD-V3-Cfg2*
- *RD-V3-Cfg1*
- *RD-V3*
- *RD-N2-Cfg3*
- *RD-N2-Cfg2*
- *RD-N2-Cfg1*
- *RD-N2*
- *RD-V2*
- *RD-V1-MC*
- *RD-V1*
- *RD-N1-Edge-X2*
- *RD-N1-Edge*
- *SGI-575*

FVP versions:

- RD-V3-R1 & RD-V3-R1-Cfg1 : 11.27.51
- RD-V3, RD-V3-Cfg1 & RD-V3-Cfg2 : 11.27.51
- RD-N2, RD-N2-Cfg1 & RD-N2-Cfg2 : 11.25.23
- RD-V2 : 11.24.12
- RD-V1 and RD-N1 variants : 11.17.29
- SGI-575 : 11.15.26

36.2 Change Log

TF-M:

- Rebased to latest main branch

SCP:

- Rebased to latest main branch
- Added compact HN table support
- Added memory region that targets GIC HNI
- Updated LCP ram size
- Fixed boot flash HNI target id in RD-V3-R1-Cfg1
- Updated DRAM2 base address in RD-V3-R1 and RD-V3-R1-Cfg1

TF-A:

- Rebased to latest main branch
- Added Local Chip Addressing (LCA) support for RD-N2-Cfg2 and RD-V3-Cfg2
- Updated console name to checksum calculation on RD-V3-R1 and RD-V3
- Enabled SMMUv3 polling timeout
- Updated DRAM2 base address in RD-V3-R1 and RD-V3-R1-Cfg1
- MbedTLS version has been updated to 3.6.2

RMM:

- Rebased to latest main branch

Hafnium:

- Kept at last release, FF-A version 1.2 is not supported by the platform

edk2:

- Rebased to latest main branch

edk2-platforms:

- Rebased to latest main branch
- Enabled support to autogenerate SoC expansion block iort table
- Added support to print Firmware Version
- Updated DRAM2 base address in RD-V3-R1 and RD-V3-R1-Cfg1

Linux:

- No updates

kvmtool and kvm-unit-tests:

- No updates

build-scripts:

- Added support to print Firmware Version in EDK2

container-scripts:

- Introduced rootless docker run feature

model-scripts:

- No updates

buildroot:

- No updates

36.3 Supported Features

- Introduced rootless docker run feature for the container environment

36.4 Known Limitations

- Virtual machine in Realm state is not booting due to a defect in RMM component.
- For RD-V3-Cfg2, boot times have increased and it is suggested to use HEADLESS mode as a workaround by using the -j option with the boot scripts. Example: *./boot.sh -p rdv3cfg2 -j*. This will not launch any UART xterm windows, but the UART logs will be captured in the log file.
- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8 GiB. Therefore, total DRAM Memory for RD-V3-Cfg2 is limited to 8 GiB to support Realm VMs and Realm KVM unit test.
- *LocateHandleBuffer_Func* tests of UEFI SCT test suite, which are executed as part of the SystemReady Compliance Program are experiencing prolonged execution times and the suite may timeout before test completion.

36.5 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V3-R1
  - Busybox boot, buildroot boot, distro boot.
- RD-V3-R1-Cfg1
  - Busybox boot, buildroot boot, distro boot.
- RD-V3-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-V3-Cfg1
  - Busybox boot, distro boot, buildroot boot, realm tests.
- RD-V3
  - Busybox boot, distro boot, buildroot boot, ACS, Virtualization.
- RD-V2
  - Busybox boot, distro boot.

- RD-N2
  - Busybox boot, distro boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot.
- RD-N2-Cfg2
  - Busybox boot, distro boot.
- RD-N2-Cfg3
  - Busybox boot, distro boot.
- RD-V1
  - Busybox boot.
- RD-V1-MC
  - Busybox boot.
- RD-N1-Edge
  - Busybox boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

36.6 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.12.20
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.12.20
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.12.20
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2024.12.20
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git

  - Tag/Hash : 41e8d5b1f805e882554b567e587c0eed5a81c49d

- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.12.20
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.12.20
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2024.12.20
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : G20240322
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-3.6.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_1
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2024.12.20
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Hash : cca/v2
- KVM unit tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests
  - Tag/Hash : RD-INFRA-2024.12.20

CHAPTER

THIRTYSEVEN

RD-INFRA-2024.09.30

37.1 Release Description

Added support for following new platforms:

- *RD-V3-R1*
- *RD-V3-R1-Cfg1*

Software stack refreshed for the following platforms:

- *RD-V3-Cfg2*
- *RD-V3-Cfg1*
- *RD-V3*
- *RD-N2-Cfg3*
- *RD-N2-Cfg2*
- *RD-N2-Cfg1*
- *RD-N2*
- *RD-V2*
- *RD-V1-MC*
- *RD-V1*
- *RD-N1-Edge-X2*
- *RD-N1-Edge*
- *SGI-575*

FVP versions:

- RD-V3-R1 & RD-V3-R1-Cfg1 : 11.27.25
- RD-V3, RD-V3-Cfg1 & RD-V3-Cfg2 : 11.27.25
- RD-N2, RD-N2-Cfg1 & RD-N2-Cfg2 : 11.25.23
- RD-V2 : 11.24.12
- RD-V1 and RD-N1 variants : 11.17.29
- SGI-575 : 11.15.26

Change logs:

TF-M:

- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.
- Configured ATU to access AP shared SRAM in RdV3.
- Added interrupt handler for SCP-RSE MHUv3.
- Added common area for Neoverse sub-platforms.
- Added BL2 config multiload support.
- Added AP reset to BL31 support.

SCP:

- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.
- Enabled warm reboot support in RdV3 platform variants.
- Added AP reset to BL31 support.

TF-A:

- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.
- Enabled warm reboot support in RdV3 platform variants.
- Added AP reset to BL31 support.

RMM:

- Rebased to latest main branch.
- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.

Hafnium:

- Kept at last release, FF-A version 1.2 is not supported by the platform.

edk2:

- Rebased to latest main branch.

edk2-platforms:

- Added new AEST node entries to AEST ACPI table to represent CMN RAS errors on RD-V3-Cfg1 platform.
- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.

Linux:

- Added support for CMN Cyprus (CMN S3) Kernel First Handling on RD-V3-Cfg1 platform. This feature can be validated only on Pre-Silicon platform. The software (linux kernel drivers, ACPI tables) are all functional.

kvmtool and kvm-unit-tests:

- No updates.

build-scripts:

- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.
- Added AP reset to BL31 support.

model-scripts:

- Added support for new platforms RD-V3-R1 and RD-V3-R1-Cfg1.
- Added AP reset to BL31 support.

buildroot:

- No updates

37.2 Supported Features

- Warm Reset support
- Reset to BL31 support

37.3 Known Limitations

- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8GiB. Therefore, total DRAM Memory for RD-V3-Cfg2 is limited to 8GiB to support Realm VMs and Realm KVM unit test.

37.4 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V3-R1
  - Busybox boot, buildroot boot, distro boot.
- RD-V3-R1-Cfg1
  - Busybox boot, buildroot boot, distro boot.
- RD-V3-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-V3-Cfg1
  - Busybox boot, distro boot, buildroot boot, realm tests.
- RD-V3
  - Busybox boot, distro boot, buildroot boot, ACS, Virtualization.
- RD-V2
  - Busybox boot, distro boot.
- RD-N2
  - Busybox boot, distro boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot.
- RD-N2-Cfg2
  - Busybox boot, distro boot.
- RD-N2-Cfg3
  - Busybox boot, distro boot.

- RD-V1
  - Busybox boot.
- RD-V1-MC
  - Busybox boot.
- RD-N1-Edge
  - Busybox boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

37.5 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.09.30
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.09.30
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.09.30
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2024.09.30
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git
  - Tag/Hash : 41e8d5b1f805e882554b567e587c0eed5a81c49d
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.09.30
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.09.30
- Linux

  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2024.09.30
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : G20240322
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-3.6.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2024.09.30
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Hash : cca/v2
- KVM unit tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests
  - Tag/Hash : RD-INFRA-2024.09.30

CHAPTER

THIRTYEIGHT

RD-INFRA-2024.07.15

38.1 Release Description

Software stack refreshed for the following platforms.

- *RD-V3-Cfg2*
- *RD-V3-Cfg1*
- *RD-V3*
- *RD-N2-Cfg3*
- *RD-N2-Cfg2*
- *RD-N2-Cfg1*
- *RD-N2*
- *RD-V2*
- *RD-V1-MC*
- *RD-V1*
- *RD-N1-Edge-X2*
- *RD-N1-Edge*
- *SGI-575*

Change logs:

TF-M:

- Renamed RdFremont to RdV3.

SCP:

- Renamed RdFremont to RdV3.

TF-A:

- Renamed RdFremont to RdV3.

RMM:

- Renamed RdFremont to RdV3.

Hafnium:

- Kept at last release, FF-A version 1.2 is not supported by the platform.

edk2:

- Rebased to latest master.

edk2-platforms:

- Added MPAM support for the RDV3 platform.
- Renamed RdFremont to RdV3.

Linux:

- Updated to CCA v3.

kvmtool and kvm-unit-tests:

- Updated to CCA v2.

build-scripts:

- Renamed all instances of RdFremont to RdV3 in config data.
- Build LKVM as a static binary to remove the dependancy on the target OS libc.

model-scripts:

- Renamed all instances of RdFremont to RdV3 in config data.
- Update run_model parameter to use iris interface for DS5 connection.

buildroot:

- Rebased to latest master.

38.2 Supported Features

MPAM:

- Added MPAM resctrl support for the RD-V3 platform. Please note that MPAM from a performance stand-point cannot be tried out on FVP. The software layers (programming schemata, discovering MSCs via ACPI) should all be functional.
- Unified MPAM support for RD-V3 and RD-N2-Cfg1 with the same kernel tag.
- MPAM kernel tag has been moved to v6.7-rc2.

38.3 Known Limitations

- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8GiB. Therefore, total DRAM Memory for RD-V3-Cfg2 is limited to 8GiB to support Realm VMs and Realm KVM unit test.
- In RD-V3-Cfg2 FVP, the peripheral base address on the remote chip's IO Block is not within the chip address space. Due to this, their NoC S3 blocks cannot be initialised. Because of this, only Chip 0's PCIe devices are enumerated and published to the OS.

38.4 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V3-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-V3-Cfg1
  - Busybox boot, distro boot, buildroot boot, realm tests.
- RD-V3
  - Busybox boot, distro boot, buildroot boot, ACS, Virtualization.
- RD-V2
  - Busybox boot, distro boot.
- RD-N2
  - Busybox boot, distro boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot.
- RD-N2-Cfg2
  - Busybox boot, distro boot.
- RD-N2-Cfg3
  - Busybox boot, distro boot.
- RD-V1
  - Busybox boot.
- RD-V1-MC
  - Busybox boot.
- RD-N1-Edge
  - Busybox boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

38.5 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.07.15
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.07.15
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.07.15
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2024.07.15
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git
  - Tag/Hash : 41e8d5b1f805e882554b567e587c0eed5a81c49d
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.07.15
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.07.15
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2024.07.15
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : G20240322
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-3.6.0

- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2024.07.15
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Hash : cca/v2
- KVM unit tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests
  - Tag/Hash : RD-INFRA-2024.07.15

CHAPTER

THIRTYNINE

RD-INFRA-2024.04.17

39.1 Release Description

Software stack refreshed for the following platforms.

- *RD-Fremont-Cfg2*
- *RD-Fremont-Cfg1*
- *RD-Fremont*
- *RD-N2-Cfg3*
- *RD-N2-Cfg2*
- *RD-N2-Cfg1*
- *RD-N2*
- *RD-V2*
- *RD-V1-MC*
- *RD-V1*
- *RD-N1-Edge-X2*
- *RD-N1-Edge*
- *SGI-575*

Change logs:

TF-M:

- RSS renamed to RSE (Runtime Security Engine).
- Fremont support patches are upstreamed.
- Checks are added to identify overlapping regions when new PSAM or APU region is added in NI-Tower driver.
- Memory maps are segregated to host_*_memory_map.h files.
- Separate SMMUv3 driver added.

SCP:

- Added support for NoC S3 (NI-Tower) driver module.
- Added support for Address remapper driver module for providing APIs capable of doing read-write operations in AP memory map.
- Introduced IO Block module for RD-Fremont platform.

- Unified PCIe Setup module for all the RD platforms.
- Enabled support for PCIe setup in RD-Fremont-Cfg2 platform.

TF-A:

- Refactored platform support for neoverse platforms in plat/arm/board/neoverse_rd/ directory.
  - Generations of nrd (Neoverse-rd):
    - nrd1, nrd_plat1 for A75/N1/V1 platforms.
    - nrd2, nrd_plat2 for N2/V2 platforms.
    - nrd3, nrd_plat3 for V3 platforms.
- Setup code common to various generations of neoverse-rd platforms moved to plat/arm/board/neoverse_rd/common/ directory.
- Platform specific setup code kept in plat/arm/board/neoverse_rd/platform/rdn2, plat/arm/board/neoverse_rd/platform/rdfremont etc.
- Added support to read SDS data in a multichip setup.
- Added support for initialising IO Block SMMUs.
- Added support for Firmware first error handling for CPU, SRAM.
- Updated CPER buffer mapping.
- Enabled logical partition support for platform with Hafnium.
- Added support to delegate RAS interrupt to Secure partition.

RMM:

- Introduced a new console library at lib/console.
- Console information is now passed in boot manifest.
- PL011 driver now uses console library APIs and work with the console info from boot manifest to initialize console.
- plat/fvp and plat/rdfremont are now merged into a common plat/arm.
- RD-Fremont config is now reduced to minimum with configuration only for RMM_MAX_SIZE and RMM_MAX_GRANULES.

Hafnium:

- Rebased to latest master.

edk2:

- Rebased to latest master.

edk2-platforms:

- Rebased to latest master.
- Enabled HEST for Firmware first error logging in kernel.
- Enabled CPU and SRAM error handling and logging in secure partition.
- Converted Einj addresses to platform specific PCDs.

Linux:

- SMMU-test-engine patches integrated on top of EAC5 branch in order to support IO-virtualization use-case.

- Enabled SDEI.
- EDAC module added.
- Enabled Arm RAS trace events.

kvmtool and kvm-unit-tests:

- Added a new script (run_tests_kvmtool_arm.sh) to run non-secure kvm-unit-tests. This script allows running non-secure kvm-unit-tests on buildroot filesystem itself without booting into a Linux distro.

build-scripts:

- Make build support on SCP build-script is deprecated.
- Additional build flag can be passed to SCP through *SCP_BUILD_FLAGS* parameter from config data.
- SCP build system defaults to Ninja.
- Improved incremental build support for SCP.
- Renamed all instances of RSS to RSE in config data.
- TF-A build configs updated to accommodate the latest refactoring.
- TF-M build script is update to accommodate different provisioning bundle per chip.
- Toolchain base path in updated from *${WORKSPACE}/tools/gcc* to *${WORKSPACE}/tools*
- Enabled io-virtualization tests on RD-N2 and RD-Fremont platforms configs.
- RAS support enabled on RD-Fremont-Cfg1 config.
- RAS daemon support enabled for RD-N2-Cfg1 and RD-Fremont-Cfg1 buildroot configs.
- Enabled build support for SBSA ACS.

model-scripts:

- RSE CM bundle load location updated for RD-Fremont variants.
- Distro support enabled for RD-Fremont-Cfg2.
- ACS support enabled for RD-Fremont.

buildroot:

- Added rasdaemon tool.

Miscellaneous:

- The documentation has been restructured for better navigation.

39.2 Supported Features

Power Management:

- Support for Shutdown, Cold and Warm reboot is added . Code changes are done in SCP, TF-M for establishing MHU outband communications between SCP-MCP and SCP-RSS to relay Shutdown/Reboot SCMI messages.
  - *Reboot-Shutdown test*
- Necessary configurations for SMCF and AMU are added in SCP. Platform SMCF and Client SMCF modules are introduced in SCP. An user control, using AP-SCP Non-Secure MHU is added. On receiving MHU signal, SMCF client module will start SMCF sampling, capture AMU data for all cores and stop sampling. In TF-A MPMM and AMU Aux counters are enabled using fconf.

  - *RdFremont SMCF*

RAS:

- Error injection from linux kernel for CPU and SRAM is supported. SRAM error, of CE type, handling happens in Root world in context of TF-A. CPU error, of type DE, can be handled either Kernel first or Firmware first manner. This RAS feature is supported only on RdFremontCfg1 and RdN2Cfg1 platform.

  A build flag TF_A_RAS_FW_FIRST is present in build-script to opt for Firmware first or kernel first mode.

  Support is added in EDK2 PlatformErrorHandlerDxe for handling Vendor specific error injection in kernel. Necessary EINJ ACPI table is added. AEST ACPI table is added for error handling in kernel.

  In Linux a new driver for handling vendor specific error injection is added and necessary modifications are made in einj driver. AEST driver is added and modification are made in linux for handling CPU Deferred Error(DE) error in kernel. In kernel, also EDAC module is added for logging CPU errors in EDAC sysfs interfaces. FTRACE is enabled in kernel to log ARM RAS traces.

  In TF-A code changes are done for enabling EHF framework, carving out region for CPER & EINJ buffers, enabling SRAM 1-bit Corrected Error(CE) injection & handling. During Firmware first handling, error is logged in CPER and using SDEI mechanism passed onto kernel.

  In Buildroot, Rasdaemon is enabled to capture Arm RAS trace events.

  - *Rdfremont RAS*

- A command line based RAS error injection and handling module is introduced in SCP. Using SCP CLI debugger interfaces, this module allows user to provide RAS error injection commands for various components: Peripheral SRAM, SCP TCM, RSM SRAM, AP core. This utility module helps in validating RAS capable hardware components' behavior when error is detected and reported.

  - *SCP RAS Error Injection Utility*

39.3 Known Limitations

- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8GiB. Therefore, total DRAM Memory for RD-Fremont-Cfg2 is limited to 8GiB to support Realm VMs and Realm KVM unit test.
- In RD-Fremont-Cfg2 FVP, the peripheral base address on the remote chip's IO Block is not within the chip address space. Due to this, their NoC S3 blocks cannot be initialised. Because of this, only Chip 0's PCIe devices are enumerated and published to the OS.

39.4 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-Fremont-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-Fremont-Cfg1
  - Busybox boot, distro boot, buildroot boot, realm tests.
- RD-Fremont

  - Busybox boot, distro boot, buildroot boot, ACS, Virtualization.
- RD-V2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot, buildroot boot, Virtualization, N2 RAS, SRAM RAS.
- RD-N2-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-N2-Cfg3
  - Busybox boot, distro boot, buildroot boot.
- RD-V1
  - Busybox boot, distro boot.
- RD-V1-MC
  - Busybox boot, distro boot.
- RD-N1-Edge
  - Busybox boot, distro boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

39.5 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.04.17
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.04.17
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.04.17
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm

  - Tag/Hash : RD-INFRA-2024.04.17
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git
  - Tag/Hash : 41e8d5b1f805e882554b567e587c0eed5a81c49d
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.04.17
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.04.17
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2024.04.17
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R09_25_20
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-3.4.1
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2024.04.17
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Hash : cca/rmm-v1.0-eac5
- KVM unit tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests

- Tag/Hash : RD-INFRA-2024.04.17

CHAPTER

FORTY

RD-INFRA-2024.01.16

40.1 Release Description

Change logs:

TF-M:

- Fremont support patches updated for upstream.
- Tower-NCI driver refactor and renamed to NI-Tower.
- SCP and MCP ATUs are configured as manage mode to let them configure respective ATU.
- With latest TF-M upstream, Non-Secure image is not build as part of TF-M.
- MHUv3 driver updated to use in-band communication using door bell channels instead of out-band communication.
- RSS to RSS communication enabled via MHUv2. RSS to RSS comms channel is used for handshaking and generating vHUK in case of multichip scenario (RD-Fremont-Cfg2).
- Reboot support added in RSS to receive and acknowledge any reboot request from SCP.

SCP:

- New product group introduced which incorporate all RD platforms. So rdfremont folder moved under product/neoverse-rd folder.
- CMN-Cyprus driver module updated match upstream revision.
- Component Port Aggregation (CPA), LCN SAM programming support included in the CMN-Cryprus driver module.
- SCP configured to manage its own ATU.
- SMCF support enabled by introducing amu_smcf_drv module.
- Reboot and Power down support enabled.
- CLI debugger enabled for RD-Fremont.
- RAS support.

TF-A:

- Poseidon VANE CPU core MIDR updated and Poseidon V CPU MIDR introduced.
- Moved away from using common arm_def.h header file to Neoverse RD specifc sgi_common_def.h header file.
- RD-Fremont variant specific CSS support files introduced which included definition for CSS and RoS address space.

- GPT setup from plat/arm/common/arm_bl2_setup.c moved to platform specific plat/arm/board/rdfremont/rdfremont_plat.c file.
- MHUv3 driver updated to support in-band communication.
- GPC SMMU block initialized for remote chips.
- Added support for Warm reboot.
- Added support for RAS EINJ.

RMM:

- RMM updated to align with RMM EAC5 specification.
- DRAM management moved to platform specific code.
- Platform setup code made common between FVP and RD-Fremont.

Hafnium:

- Latest upstream change removed clang toolchain from prebuilds. Clang toolchain need to be passed via $PATH environment variable.
- Hafnium builds now needs platform config name to be passed while invoking build.

edk2:

- EINJ specific structures introduced to ACPI header files.

edk2-platforms:

- Reduced PcdSystemMemorySize to accommodate growing needs to EL3 runtime and RMM.
- EINJ and AEST ACPI tables added for RD-Fremont-Cfg1.

Linux:

- Kernel updated to align with align with RMM EAC5 specification.
- AEST ACPI table parser support added.
- Support for vendor defined error injection mechanism added.

kvmtool and kvm-unit-tests:

- Update to align with RMM EAC5 specification.

build-scripts:

- TF-M Non-Secure image package is skipped to align with upstream TF-M change.
- TF-M Chip Manufacturing bundle is packaged on per chip basis.
- SCP build-scripts update to support product group (neoverse-rd).
- clang+llvm-15.0.6 toolchain added as dependency to support hafnium build.
- Toolchain upgraded from GCC 12.3 Rel1 to 13.2 Rel1.
- build-linux updated to support building debian packages. Respective dependency added to install prerequisties.
- RAS EINJ, Kernel First error injection and handling support enabled for RD-Fremont-Cfg1 config.

model-scripts:

- Load different CM provisioning bundle on per chip basis.
- Updated RSS to RSS MHUv2 doorbell channel count to 5 to support in-band communication.
- Updated AP to RSS MHUv3 doorbell channel count to 16 to support in-band communication.

- Enabled SMCF tag length input.
- Added shutdown string for MCP. Once this string is printed in MCP console, model will quit gracefully.

busybox:

- Upgraded to version 1.36.0

buildroot:

- Upgraded to latest master to include support for GCC 13.2 Rel support.

Miscellaneous:

- The documentation has been migrated to use the 'readthedocs' rendering syntax. So it would be essential to setup a readthedocs server to use the links to navigate the various pages in the documentation.

40.2 Supported Features

Power Management:

- Support for Shutdown, Cold and Warm reboot is added . Code changes are done in SCP, TF-M for establishing MHU outband communications between SCP-MCP and SCP-RSS to relay Shutdown/Reboot SCMI messages.
  - *Reboot-Shutdown test*
- Necessary configurations for SMCF and AMU are added in SCP. Platform SMCF and Client SMCF modules are introduced in SCP. An user control, using AP-SCP Non-Secure MHU is added. On receiving MHU signal, SMCF client module will start SMCF sampling, capture AMU data for all cores and stop sampling.

  RdFremont FVP is enabled with tag_length support for SMCF sample. It needs model parameter to enable tag length, necessary model script change is added.
  - *RdFremont SMCF*

RAS:

- Error injection from linux kernel Non-Secure world for CPU and SRAM is supported. SRAM error, of CE type, handling happens in Root world in context of TF-A. CPU error, of type DE, can be handled either Kernel first or Firmware first manner. This RAS feature is supported only on RdFremontCfg1 platform.

  A build flag TF_A_RAS_FW_FIRST is present in build-script to opt for Firmware first or kernel first mode. Support is added in EDK2 PlatformErrorHandlerDxe for handling Vendor specific error injection in kernel. Necessary EINJ ACPI table is added. AEST ACPI table is added for error handling in kernel. In Linux a new driver for handling vendor specific error injection is added and necessary modifications are made in einj driver. AEST driver is added and modification are made in linux for handling CPU Deferred Error(DE) error in kernel. In TF-A code changes are done for enabling EHF framework, carving out region for CPER & EINJ buffers, enabling SRAM 1-bit Corrected Error(CE) injection & handling.
  - *Rdfremont RAS*
- A command line based RAS error injection and handling module is introduced in SCP. Using SCP CLI debugger interfaces, this module allows user to provide RAS error injection commands for various components: Peripheral SRAM, SCP TCM, RSM SRAM, AP core. This utility module helps in validating RAS capable hardware components' behavior when error is detected and reported.
  - *SCP RAS Error Injection Utility*

40.3 Known Limitations

- AArch64 host native build doesn't support launch of virtual machine and kvm unit test in realm due to missing library dependency in buildroot. Boot to shell of busybox and buildroot is supported.
- Current RMM release does not support creating Granules beyond 8GiB. Therefore, total DRAM Memory for RD-Fremont-Cfg2 is limited to 8GiB to support Realm VMs and Realm KVM unit test.

40.4 Test Coverage

The following tests have been completed using 11.24.16 version of the FVP:

- RD-Fremont
  - Busybox boot, distro boot, buildroot boot, secure boot, virtual machine and kvm unit test in realm.
- RD-Fremont-Cfg1
  - Busybox boot, distro boot, buildroot boot, secure boot, virtual machine and kvm unit test in realm.
  - Feature test:
    - *CPPC*
    - *Reboot-Shutdown test*
    - *RdFremont SMCF test*
    - *Rdfremont RAS related test*
    - *SCP RAS Error injection utility*
- RD-Fremont-Cfg2
  - Busybox boot, buildroot boot, virtual machine and kvm unit test in realm.

40.5 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2024.01.16
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2024.01.16
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2024.01.16
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm

  - Tag/Hash : RD-INFRA-2024.01.16
- Hafnium
  - Source : https://git.trustedfirmware.org/hafnium/hafnium.git
  - Tag/Hash : 9681574575c02764ff85b4c0903ab61a6327ed16
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2024.01.16
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2024.01.16
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2024.01.16
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R06_28_23
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://github.com/buildroot/buildroot
  - Tag/Hash : 3865d88423c18f28f74efd9878a386db9491246f
- KVM tool
  - Source : https://git.gitlab.arm.com/linux-arm/kvmtool-cca
  - Tag/Has : cca/rmm-v1.0-eac5
- KVM unit tests
  - Source : https://git.gitlab.arm.com/linux-arm/kvm-unit-tests-cca

- Tag/Has : cca/rmm-v1.0-eac5

CHAPTER

FORTYONE

RD-INFRA-2023.12.22

41.1 Release Description

- Software stack refreshed for the following platforms.
  - *SGI-575*
  - *RD-N1-Edge*
  - *RD-N1-Edge-x2*
  - *RD-V1*
  - *RD-V1-MC*
  - *RD-N2*
  - *RD-N2-Cfg1*
  - *RD-N2-Cfg2*
  - *RD-N2-Cfg3*
  - *RD-V2*

41.2 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot, buildroot boot, Virtualization, N2 RAS, SRAM RAS.
- RD-N2-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-N2-Cfg3
  - Busybox boot, distro boot, buildroot boot.

- RD-V1
  - Busybox boot, distro boot.
- RD-V1-MC
  - Busybox boot, distro boot.
- RD-N1-Edge
  - Busybox boot, distro boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

41.3 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- SCP Firmware
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware.git
  - Tag/Hash : RD-INFRA-2023.12.22
- Trusted Firmware-A
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a.git
  - Tag/Hash : RD-INFRA-2023.12.22
- EDK2
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2.git
  - Tag/Hash : RD-INFRA-2023.12.22
- EDK2 Platforms
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms.git
  - Tag/Hash : RD-INFRA-2023.12.22
- Linux
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/linux.git
  - Tag/Hash : RD-INFRA-2023.12.22
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R06_28_23
- Mbed TLS

  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot.git
  - Tag/Hash : RD-INFRA-2023.12.22
- kvmtool
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool
  - Tag/Hash : e17d182ad3f797f01947fc234d95c96c050c534b
- kvm-unit-tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests.git
  - Tag/Hash : RD-INFRA-2023.12.22

CHAPTER

FORTYTWO

RD-INFRA-2023.09.29

42.1 Release Description

- Software stack refreshed for the following platforms.
  - *SGI-575*
  - *RD-N1-Edge*
  - *RD-N1-Edge-x2*
  - *RD-V1*
  - *RD-V1-MC*
  - *RD-N2*
  - *RD-N2-Cfg1*
  - *RD-N2-Cfg2*
  - *RD-N2-Cfg3*
  - *RD-V2*
- Platform software stack build updated to use Arm GCC toolchain version 12.3.rel1

42.2 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, secure boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot, buildroot boot, Virtualization, N2 RAS, SRAM RAS.
- RD-N2-Cfg2
  - Busybox boot, distro boot, buildroot boot.
- RD-N2-Cfg3

  - Busybox boot, distro boot, buildroot boot.
- RD-V1
  - Busybox boot, distro boot.
- RD-V1-MC
  - Busybox boot, distro boot.
- RD-N1-Edge
  - Busybox boot, distro boot.
- RD-N1-Edge-X2
  - Busybox boot.
- SGI-575
  - Busybox boot.

42.3 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- SCP Firmware
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware.git
  - Tag/Hash : RD-INFRA-2023.09.29
- Trusted Firmware-A
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a.git
  - Tag/Hash : RD-INFRA-2023.09.29
- EDK2
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2.git
  - Tag/Hash : RD-INFRA-2023.09.29
- EDK2 Platforms
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms.git
  - Tag/Hash : RD-INFRA-2023.09.29
- Linux
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/linux.git
  - Tag/Hash : RD-INFRA-2023.09.29
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R06_28_23

- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot.git
  - Tag/Hash : RD-INFRA-2023.09.29
- kvmtool
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool
  - Tag/Hash : e17d182ad3f797f01947fc234d95c96c050c534b
- kvm-unit-tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests.git
  - Tag/Hash : RD-INFRA-2023.09.29

CHAPTER

FORTYTHREE

RD-INFRA-2023.09.28

43.1 Release Description

– Release introduces changes to the following platforms

- *RD-Fremont*
- *RD-Fremont-Cfg1*
- *RD-Fremont-Cfg2*

– Changes introduced in this release

- Introduce CCA CoT support in TF-A
- Updates to RMM, Linux and KVM tools to align to the RMM EAC2 specification
- Introduce support for Hafnium in RD-Fremont and RD-Fremont-Cfg1
- Enable Secure Boot support for RD-Fremotn and RD-Fremont-Cfg1
- Add support for NI-Tower in SCP
- Enable configuring IO Virtualization block with NI-Tower driver in SCP
- Enable Dynamic PCIe support
- Add alpha support for Component Port Aggregation (CPA) in CMN-Cyprus driver
- Add alpha support for Expanded RAID in CMN-Cyprus driver
- Add support for configuring the GPC SMMU (System TCU+TBU)
- Enable support for Branch Record Buffer Extension (BRBE)
- Update software compoenents to latest upstream

43.2 Known Limitations

- Hafnium is not enabled for RD-Fremont-Cfg2

43.3 Test Coverage

The following tests have been completed using 11.23.11 version of the FVP:

- RD-Fremont
  - Busybox boot, distro boot, buildroot boot, secure boot, virtual machine and kvm unit test in realm.
- RD-Fremont-Cfg1
  - Busybox boot, distro boot, buildroot boot, secure boot, virtual machine and kvm unit test in realm.
- RD-Fremont-Cfg2
  - Busybox boot, buildroot boot, virtual machine and kvm unit test in realm.

43.4 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2023.09.28
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2023.09.28
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2023.09.28
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2023.09.28
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2023.09.28
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2023.09.28
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2023.09.28
- Grub
  - Source : https://git.savannah.gnu.org/git/grub

  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R06_28_23
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_36_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot
  - Tag/Hash : RD-INFRA-2023.09.28

CHAPTER

FORTYFOUR

RD-INFRA-2023.06.30

44.1 Release Description

- Software stack refreshed for the following platforms.
  - *SGI-575*
  - *RD-N1-Edge*
  - *RD-N1-Edge-x2*
  - *RD-V1*
  - *RD-V1-MC*
  - *RD-N2*
  - *RD-N2-Cfg1*
  - *RD-N2-Cfg2*
  - *RD-N2-Cfg3*
  - *RD-V2*

44.2 Test Coverage

The following tests have been completed for this release. The FVP version used is platform specific and can be found in the in the release tags section of the platform readme.

- RD-V2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, linuxboot, secure boot.
- RD-N2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, linuxboot, secure boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot, buildroot boot, Virtualization, N2 RAS, SRAM RAS, tf-a-tests, linuxboot, secure boot.
- RD-N2-Cfg2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization.

- RD-V1
  - Busybox boot, distro boot, UEFI secure boot.
- RD-V1-MC
  - Busybox boot, distro boot, UEFI secure boot.
- RD-N1-Edge
  - Busybox boot, distro boot.
- RD-N1-Edge-X2
  - Busybox boot, distro boot.
- SGI-575
  - Busybox boot, distro boot.

44.3 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- SCP Firmware
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware.git
  - Tag/Hash : RD-INFRA-2023.06.30
- Trusted Firmware-A
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a.git
  - Tag/Hash : RD-INFRA-2023.06.30
- EDK2
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2.git
  - Tag/Hash : RD-INFRA-2023.06.30
- EDK2 Platforms
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms.git
  - Tag/Hash : RD-INFRA-2023.06.30
- Linux
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/linux.git
  - Tag/Hash : RD-INFRA-2023.06.30
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R09_25_20
- Mbed TLS

  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_33_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot.git
  - Tag/Hash : RD-INFRA-2023.03.31
- kvmtool
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool
  - Tag/Hash : 95f47968a1d34ea27d4f3ad767f0c2c49f2ffc5b
- kvm-unit-tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests.git
  - Tag/Hash : RD-INFRA-2023.03.31

CHAPTER

FORTYFIVE

RD-INFRA-2023.06.28

45.1 Release Description

– Release introduces changes to the following platforms

- *RD-Fremont*
- *RD-Fremont-Cfg1*
- *RD-Fremont-Cfg2*

– Changes introduced in this release

- Introduce support for RD-Fremont-Cfg2 (quad-chip) platform
- Introduce beta support for RME
- Introduce beta support for Measured Boot in TF-M and TF-A
- Enable BL1->BL2 based boot flow
- Add support for NI-Tower in TF-M
- Add support for HN-S Isolation feature in CMN-Cyprus driver
- Add support for Bypass Discovery feature in CMN-Cyprus driver

45.2 Known Limitations

- System TCU+TBU is not present in the 11.22.16 version of the FVP. So GPC with System TCU+TBU is not enabled in the software.

45.3 Test Coverage

The following tests have been completed using 11.22.16 version of the FVP:

- RD-Fremont
  - Busybox boot, distro boot, buildroot boot.
- RD-Fremont-Cfg1
  - Busybox boot, distro boot, buildroot boot.
- RD-Fremont-Cfg2

  - Busybox boot, buildroot boot.

45.4 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2023.06.28
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2023.06.28
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2023.06.28
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2023.06.28
- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2023.06.28
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2023.06.28
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2023.06.28
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R09_25_20
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox

  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_33_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://github.com/buildroot/buildroot
  - Tag/Hash : 2023.02

CHAPTER

FORTYSIX

RD-INFRA-2023.03.31

46.1 Release Description

- Platform software stack hosting migrated from https://gitlab.arm.com/arm-reference-solutions to https://gitlab.arm.com/infra-solutions/reference-design. Previous releases have to be accessed from the previous hosting location.
- Software stack refreshed for the following platforms.
  - *SGI-575*
  - *RD-N1-Edge*
  - *RD-N1-Edge-x2*
  - *RD-V1*
  - *RD-V1-MC*
  - *RD-N2*
  - *RD-N2-Cfg1*
  - *RD-N2-Cfg2*
  - *RD-N2-Cfg3*
  - *RD-V2*

46.2 Test Coverage

The following tests have been completed using 11.20.18 version of the FVP.

- RD-V2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, linuxboot, secure boot.
- RD-N2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization, tf-a-tests, linuxboot, secure boot.
- RD-N2-Cfg1
  - Busybox boot, distro boot, buildroot boot, Virtualization, N2 RAS, SRAM RAS, tf-a-tests, linuxboot, secure boot.

- RD-N2-Cfg2
  - Busybox boot, distro boot, buildroot boot, WinPE boot, ACS, Virtualization.
- RD-V1
  - Busybox boot, distro boot, UEFI secure boot.
- RD-V1-MC
  - Busybox boot, distro boot, UEFI secure boot.
- RD-N1-Edge
  - Busybox boot, distro boot.
- RD-N1-Edge-X2
  - Busybox boot, distro boot.
- SGI-575
  - Busybox boot, distro boot.

46.3 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- SCP Firmware
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware.git
  - Tag/Hash : RD-INFRA-2023.03.31
- Trusted Firmware-A
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a.git
  - Tag/Hash : RD-INFRA-2023.03.31
- EDK2
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2.git
  - Tag/Hash : RD-INFRA-2023.03.31
- EDK2 Platforms
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms.git
  - Tag/Hash : RD-INFRA-2023.03.31
- Linux
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/linux.git
  - Tag/Hash : RD-INFRA-2023.03.31
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica

  - Tag/Hash : R09_25_20
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_33_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot.git
  - Tag/Hash : https://git.gitlab.arm.com/infra-solutions/reference-design/platsw/buildroot.git
- kvmtool
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool
  - Tag/Hash : 95f47968a1d34ea27d4f3ad767f0c2c49f2ffc5b
- kvm-unit-tests
  - Source : https://git.gitlab.arm.com/infra-solutions/reference-design/valsw/kvm-unit-tests.git
  - Tag/Hash : RD-INFRA-2023.03.31

CHAPTER

FORTYSEVEN

RD-INFRA-2023.03.29

47.1 Release Description

- Introduce support for *RD-Fremont* and *RD-Fremont-Cfg1* platforms.

47.2 Test Coverage

The following tests have been completed using 11.21.18 version of the FVP:

- RD-Fremont
  - Busybox boot distro boot, buildroot boot.
- RD-Fremont-Cfg1
  - Busybox boot distro boot, buildroot boot.

47.3 Source Repositories

The following source repositories have been integrated together in this release. The associated tag or the hash in each of these repositories is listed as well.

- Trusted Firmware-M
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-m
  - Tag/Hash : RD-INFRA-2023.03.29
- SCP Firmware
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/scp-firmware
  - Tag/Hash : RD-INFRA-2023.03.29
- Trusted Firmware-A
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/trusted-firmware-a
  - Tag/Hash : RD-INFRA-2023.03.29
- Trusted Firmware-RMM
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/tf-rmm
  - Tag/Hash : RD-INFRA-2023.03.29

- EDK2
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2
  - Tag/Hash : RD-INFRA-2023.03.29
- EDK2 Platforms
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/edk2-platforms
  - Tag/Hash : RD-INFRA-2023.03.29
- Linux
  - Source : https://gitlab.arm.com/infra-solutions/reference-design/platsw/linux
  - Tag/Hash : RD-INFRA-2023.03.29
- Grub
  - Source : https://git.savannah.gnu.org/git/grub
  - Tag/Hash : grub-2.04
- ACPICA
  - Source : https://github.com/acpica/acpica
  - Tag/Hash : R09_25_20
- Mbed TLS
  - Source : https://github.com/ARMmbed/mbedtls.git
  - Tag/Hash : mbedtls-2.28.0
- Busybox
  - Source : https://github.com/mirror/busybox
  - Tag/Hash : 1_33_0
- EFI Tools
  - Source : https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools
  - Tag/Hash : v1.9.2
- Buildroot
  - Source : https://github.com/buildroot/buildroot
  - Tag/Hash : 2020.05