Neoverse Reference Design Platform Software

unknown

Jan 17, 2024

CONTENTS

Note: Infrastructure reference design documentation is now available via a Read the Docs (RTD) instance. This provides a rendered view which provides a navigation menu and supports hyperlinks between documentation sections.

CHAPTER

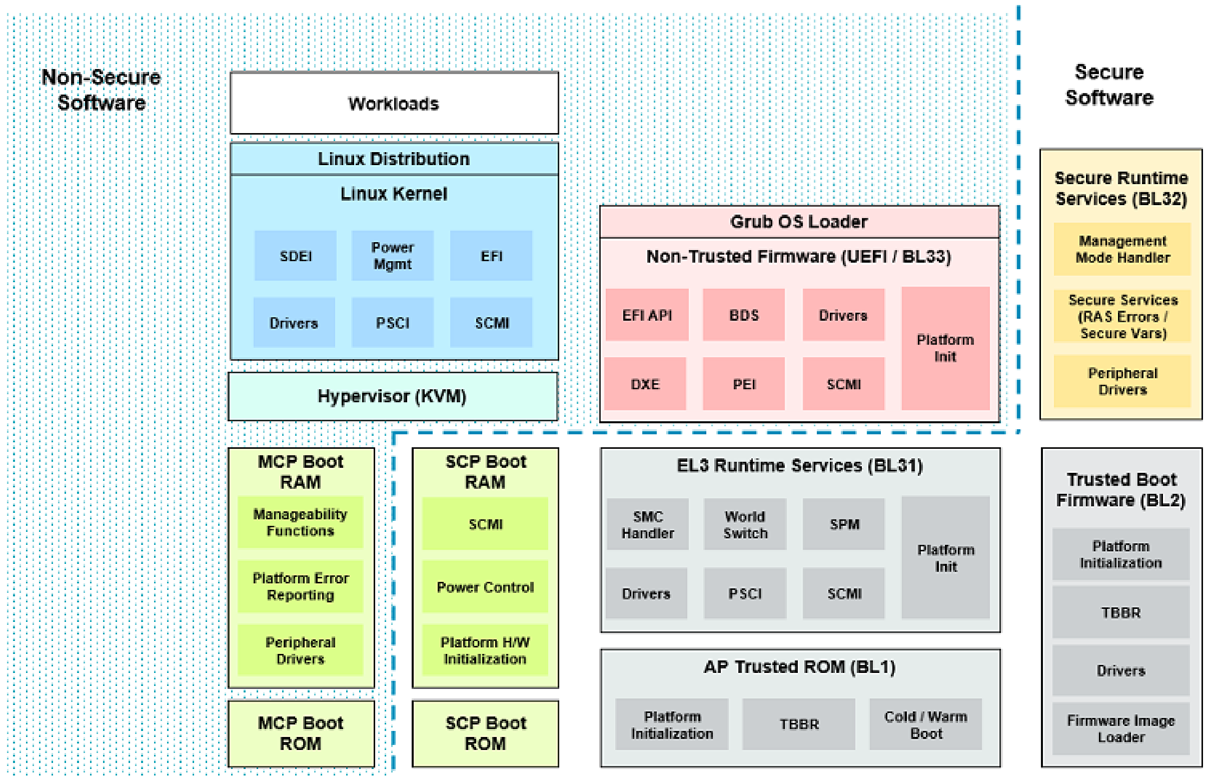
ONE

NEOVERSE REFERENCE DESIGN PLATFORM SOFTWARE

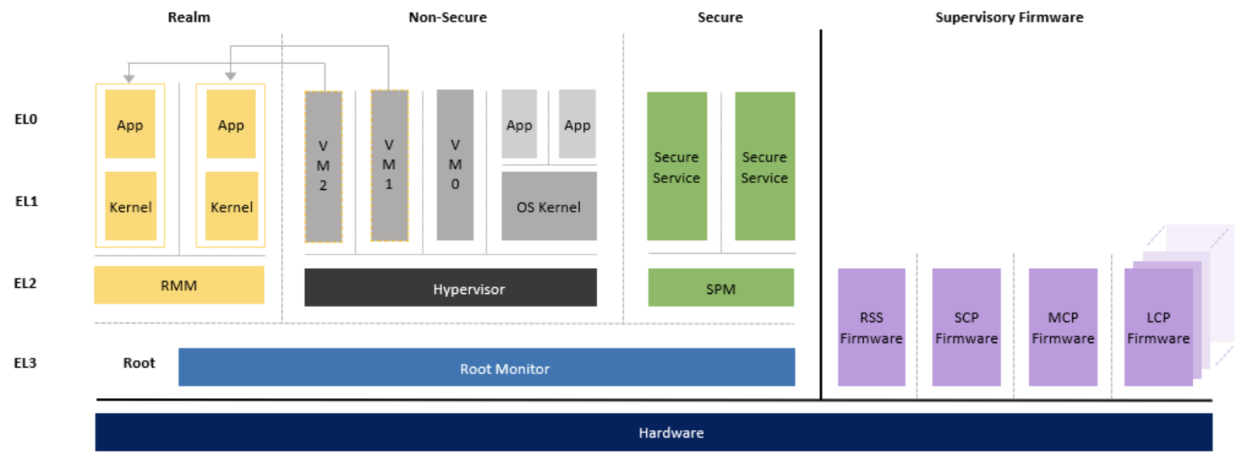
Note: Infrastructure reference design documentation is now available via a Read the Docs (RTD) instance. This provides a rendered view which provides a navigation menu and supports hyperlinks between documentation sections.

Neoverse reference designs provide useful 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 specific applications in the cloud-to-edge infrastructure markets.

The Neoverse reference designs are also available as fixed virtual platform models and provides a platform to explore various aspects of the compute subsystem design from a software perspective. A software stack is also made available as part of the Neoverse reference designs. The figure below provides a high-level representation of the Neoverse Reference Design platform solutions stack.



For platforms that support the Realm Management Extension (RME), the figure below provides a high-level representation of the Neoverse Reference Design platform solutions stack.



The platform software stack that can be used along with the fixed virtual platforms is available for the following reference design platforms.

The code supporting these platforms is available here.

- *RD-Fremont*
- *RD-Fremont-Cfg1*
- *RD-Fremont-Cfg2 (quad chip)*
- *RD-V2*
- *RD-N2*
- *RD-N2-Cfg1*
- *RD-N2-Cfg2 (quad chip)*
- *RD-N2-Cfg3 (Genesis N2 Chiplet)*
- *RD-V1 (single chip)*
- *RD-V1 (quad chip)*
- *RD-N1-Edge (single chip)*
- *RD-N1-Edge (dual chip)*
- *SGI-575* (former system guidance platform)

The links above provide detailed documentation about the platform documentation and includes instructions on downloading the software stack, compiling it, executing the software stack on a fixed virtual platform and information about the various features supported by the software stack.

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

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.

For reporting security vulnerabilities, please refer *Vulnerability reporting* page.

Copyright (c) 2021-2024, Arm Limited. All rights reserved.

CHAPTER TWO

SUPPORTED PLATFORMS

Currently, the following platforms are supported.

- RD-Fremont
- RD-Fremont Cfg1
- RD-Fremont Cfg2
- RD-V2
- RD-N2 Cfg3
- RD-N2 Cfg2
- RD-N2 Cfg1
- RD-N2
- RD-V1 Multichip
- RD-V1
- RD-N1-Edge Dual-Chip
- RD-N1-Edge
- SGI-575

2.1 RD-Fremont platform software user guide

2.1.1 Introduction

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-Fremont 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 Subsystem (RSS) 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-Fremont config supports 16xMP1 Neoverse Poseidon-V CPUs.

This document is a user guide on how to setup, build and run the RD-Fremont software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page. An overview of the platform's boot flow is described in *boot flow overview* page.

2.1.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.1.3 Obtaining the RD-Fremont FVP

The latest version of the RD-Fremont 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 Fremont Reference Design FVP" link to obtain the list of available Neoverse RD-Fremont Reference Design FVPs.
- Select a FVP build under the section "Download RD-Fremont" 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-Fremont FVP executable is included in the downloaded installer and named as "FVP_RD_Fremont". 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.

2.1.4 Supported Features by RD-Fremont platform software stack

RD-Fremont platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *UEFI Secure Boot*
- *Realm management extension*
- *Rdfremont RAS*
  - *RdFremont CPU RAS support*
  - *Shared RAM ECC RAS support*
  - *SCP RAS Error injection utility*

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

2.1.5 Miscellaneous Documentation Links

Documentation covering some of the implementation and integration aspects are listed below.

- *Boot flow*
- *RSS*
- *NI-Tower Overview*
- System Control NI-Tower
- *CMN Cyprus SCP driver*
- *Local Control Processor*
- *MCU Boot*
- *SCP ATU Configuration*
- *SCP RSS Communication*
- *AP - RSS Attestation Service*
- *Chain of Trust (CoT) for CCA*

2.1.6 Release Tags

Table below lists the release tags for the RD-Fremont platform software stack and the corresponding RD-Fremont 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-Fremont FVP Version
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

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.2 RD-Fremont-Cfg1 platform software user guide

2.2.1 Introduction

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-Fremont-Cfg1 is the first RD platform with Realm Management Extension (RME) support 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 Subsystem (RSS) 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)

This document is a user guide on how to setup, build and run the RD-Fremont-Cfg1 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page. An overview of the platform's boot flow is described in *boot flow overview* page.

2.2.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.2.3 Obtaining the RD-Fremont-Cfg1 FVP

The latest version of the RD-Fremont-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 Fremont Reference Design FVP" link to obtain the list of available Neoverse RD-Fremont-Cfg1 Reference Design FVPs.
- Select a FVP build under the section "Download RD-Fremont-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-Fremont-Cfg1 FVP executable is included in the downloaded installer and named as "FVP_RD_Fremont_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.

2.2.4 Supported Features by RD-Fremont-Cfg1 platform software stack

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

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *UEFI Secure Boot*
- *Realm management extension*

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

2.2.5 Miscellaneous Documentation Links

Documentation covering some of the implementation and integration aspects are listed below.

- *Boot flow*
- *RSS*
- *NI-Tower Overview*
- System Control NI-Tower
- *CMN Cyprus SCP driver*
- *Local Control Processor*
- *MCU Boot*
- *SCP ATU Configuration*
- *SCP RSS Communication*
- *AP - RSS Attestation Service*

2.2.6 Release Tags

Table below lists the release tags for the RD-Fremont platform software stack and the corresponding RD-Fremont 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-Fremont-Cfg1 FVP Version
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

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.3 RD-Fremont-Cfg2 platform software user guide

2.3.1 Introduction

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-Fremont-Cfg2 platform is a quad chip variant of the *RD-Fremont* platform.

RD-Fremont-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-Fremont-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 Subsystem (RSS) 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-Fremont-Cfg2 config supports quad chip with 4xMP1 Neoverse Poseidon-V CPUs per chip.

This document is a user guide on how to setup, build and run the RD-Fremont-Cfg2 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page. An overview of the platform's boot flow is described in *boot flow overview* page.

2.3.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.3.3 Obtaining the RD-Fremont-Cfg2 FVP

The latest version of the RD-Fremont-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 Fremont Reference Design FVP" link to obtain the list of available Neoverse RD-Fremont Reference Design FVPs.
- Select a FVP build under the section "Download RD-Fremont" 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-Fremont-Cfg2 FVP executable is included in the downloaded installer and named as "FVP_RD_Fremont_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.

2.3.4 Supported Features by RD-Fremont-Cfg2 platform software stack

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

- *Busybox Boot*
- *Buildroot boot*
- *Realm management extension*

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

2.3.5 Miscellaneous Documentation Links

Documentation covering some of the implementation and integration aspects are listed below.

- *Single Chip Boot flow*
- *Multichip Boot flow*
- *RSS*
- *MCU Boot*
- *CMN Cyprus SCP driver*
- *CMN-Cyprus Multichip Configuration*
- *SCP ATU Configuration*
- *SCP RSS Communication*
- *Local Control Processor*
- *RD-Fremont-Cfg2 Multichip memory map*

2.3.6 Release Tags

Table below lists the release tags for the RD-Fremont-Cfg2 platform software stack and the corresponding RD-Fremont-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-Fremont-Cfg1 FVP Version
RD-INFRA-2024.01.16	11.24.16
RD-INFRA-2023.09.28	11.23.11
RD-INFRA-2023.06.28	11.22.16

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.4 RD-V2 platform software user guide

2.4.1 Introduction

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.

This document is a user guide on how to setup, build and run the RD-V2 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.4.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.4.3 Obtaining the RD-V2 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.

2.4.4 Supported Features by RD-V2 platform software stack

RD-V2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *ACS Compliance Test*
- *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*

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.

2.4.5 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-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

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

2.5 RD-N2 Cfg3 platform software user guide

2.5.1 Introduction

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)

This document is a user guide on how to setup, build and run the RD-N2 Cfg3 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.5.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.5.3 Obtaining the RD-N2-Cfg3 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.

2.5.4 Supported Features by RD-N2-Cfg3 platform software stack

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

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *ACS Compliance Test*
- *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.

2.5.5 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-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

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

2.6 RD-N2 Cfg2 platform software user guide

2.6.1 Introduction

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 quad-chip variant of the the *RD-N2* platform) will henceforth be referred as RD-N2-Cfg2.

RD-N2-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-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)

This document is a user guide on how to setup, build and run the RD-N2-Cfg2 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.6.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.6.3 Obtaining the RD-N2-Cfg2 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.

2.6.4 Supported Features by RD-N2-Cfg2 platform software stack

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

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*

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.

2.6.5 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-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

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

2.7 RD-N2 Cfg1 platform software user guide

2.7.1 Introduction

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)

This document is a user guide on how to setup, build and run the RD-N2 Cfg1 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.7.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.7.3 Obtaining the RD-N2-Cfg1 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.

2.7.4 Supported Features by RD-N2-Cfg1 platform software stack

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*
- *RAS*
  - *N2 CPU RAS*
  - *Secure Base Element SRAM RAS*
- *Trusted Firmware-A Tests* [1]
- *Virtio-P9*
- *MPAM-resctrl*

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.

2.7.5 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-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

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

2.8 RD-N2 platform software user guide

2.8.1 Introduction

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.

This document is a user guide on how to setup, build and run the RD-N2 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.8.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.8.3 Obtaining the RD-N2 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.

2.8.4 Supported Features by RD-N2 platform software stack

RD-N2 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *ACS Compliance Test*
- *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*

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.

2.8.5 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-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

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.9 RD-V1 Quad-Chip platform software user guide

2.9.1 Introduction

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.

The "RD-V1 Quad-Chip" platform will be referred to as "RD-V1-MC" henceforth in this document. 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.

This document is a user guide on how to setup, build and run the RD-V1-MC software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.9.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.9.3 Obtaining the RD-V1-MC 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.

2.9.4 Supported Features by RD-V1-MC platform software stack

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

- *Busybox Boot*.
- *Linux Distribution Boot*.
- *UEFI Secure Boot*.

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

2.9.5 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-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

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.10 RD-V1 platform software user guide

2.10.1 Introduction

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.

RD-V1 also comes in a multi-chip variant called RD-V1-MC. The user-guide for RD-V1-MC can be obtained form *here*

This document is a user guide on how to setup, build and run the RD-V1 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.10.2 Setting up the Host Machine and Syncing the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.10.3 Obtaining the RD-V1 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.

2.10.4 Supported Features by RD-V1 platform software stack

RD-V1 platform software stack supports the following features.

- *Busybox Boot*
- *Buildroot boot*
- *Linux Distribution Boot*
- *ACS Compliance Test*
- *UEFI Secure Boot*
- *Virtualization*
- *Trusted Firmware-A Tests*

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

2.10.5 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-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

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.11 RD-N1-Edge Dual-Chip platform software user guide

2.11.1 Introduction

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.

This document is a user guide on how to setup, build and run the RD-N1-Edge-Dual software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.11.2 Setting up the Host Machine and downloading the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on '*Setup Workspace*' page.

2.11.3 Obtaining the RD-N1-Edge-Dual 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.

2.11.4 Supported features

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

- *Busybox Boot*.

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

2.11.5 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-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

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

2.12 RD-N1-Edge platform software user guide

2.12.1 Introduction

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)

This document is a user guide on how to setup, build and run the RD-N1-Edge software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.12.2 Setting up the Host Machine and downloading the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on *Setup Workspace* page.

2.12.3 Obtaining the RD-N1-Edge 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.

2.12.4 Supported features

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

- *Busybox Boot*.
- *Linux Distribution Boot*

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

2.12.5 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-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

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

2.13 SGI-575 platform software user guide

2.13.1 Introduction

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)

This document is a user guide on how to setup, build and run the SGI-575 software stack on the Fixed Virtual Platform. The components integrated into this stack is described in the *software stack* introduction page.

2.13.2 Setting up the Host Machine and downloading the software stack

Host Machine requirements and the necessary steps to be followed to sync the software stack are listed on *Setup Workspace* page.

2.13.3 Obtaining the SGI-575 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.

2.13.4 Supported features

SGI-575 platform software stack supports the following features.

- *Busybox Boot*

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

2.13.5 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-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

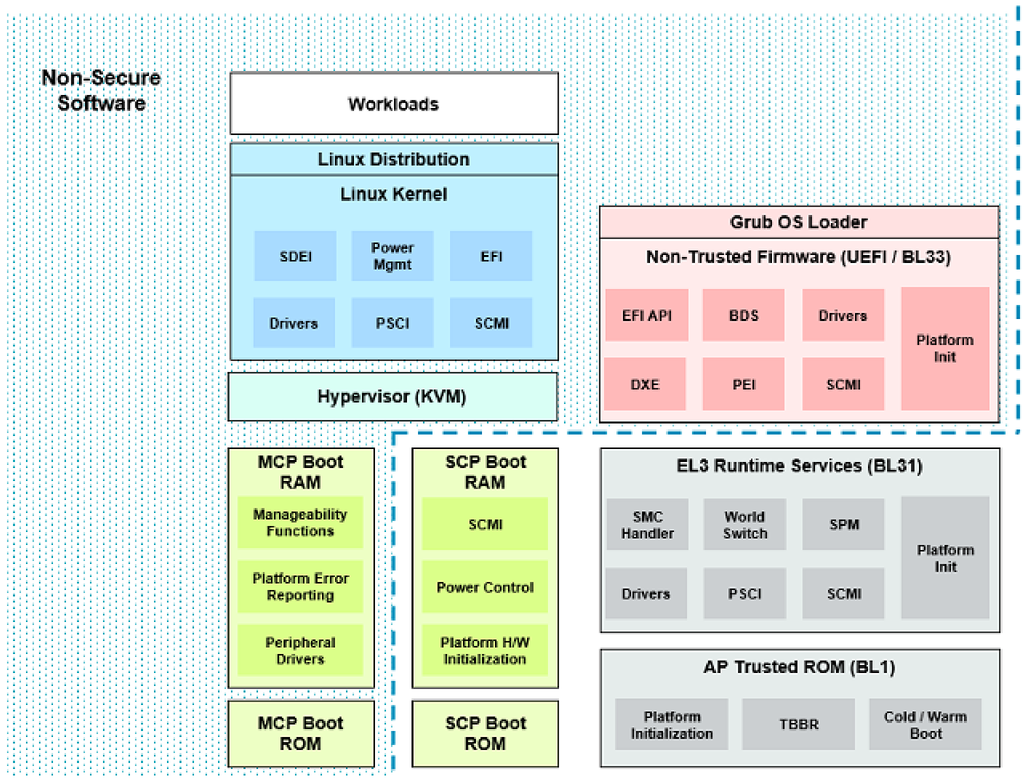
Copyright (c) 2021-2023, Arm Limited. All rights reserved.

2.14 Miscellaneous Platform Documents

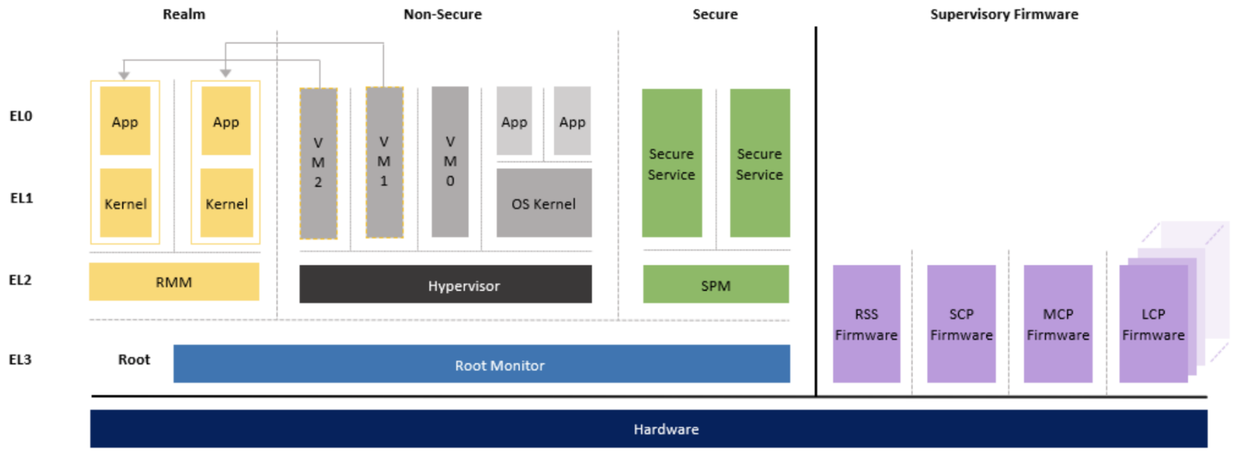
2.14.1 Neoverse Reference Design Platform Solution Stack

Introduction

The Neoverse Reference Design platform solutions stack integrates multiple software components to provide a reference implementation of software solution that can be used demonstrate various capabilities of the platform and the software stack. The figure below provides a high-level representation of the Neoverse Reference Design platform solutions stack.



The following figure provides a high-level representation of the Neoverse Reference Design platforms that support Realm Management Extension (RMM).



The following sections provide an overview of the various software components that are included the Neoverse Reference Design platform solutions stack.

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.

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.

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.

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.

Other software components

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

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

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.14.2 Platform Names

Arm's Neoverse reference design platforms are assigned names in order to allow the build and execute scripts to recognize the platform for which the software has to be built and executed. The names for the reference platforms are listed below. Please make a note of the name of the platform of your choice. This name is required to start the build and execute procedure. The names under 'Platform Name' column has to be used in place of the placeholder `<platform>` as mentioned in the commands in other documents.

Reference Platform	Platform Name
RD-Fremont	rdfremont
RD-Fremont-Cfg1	rdfremontcfg1
RD-Fremont-Cfg2	rdfremontcfg2
RD-V2	rdv2
RD-N2	rdn2
RD-N2-Cfg1	rdn2cfg1
RD-N2-Cfg2 (Quad chip)	rdn2cfg2
RD-N2-Cfg3	rdn2cfg3
RD-V1 (Single Chip)	rdv1
RD-V1 (Quad Chip)	rdv1mc
RD-N1-Edge (Single Chip)	rdn1edge
RD-N1-Edge (Dual Chip)	rdn1edgex2
SGI-575	sgi575

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.14.3 Setup the Neoverse Reference Design software stack workspace

Introduction

The page describes the procedure to sync (download) Arm's Neoverse Reference Design (RD) platform software stack.

Note: AArch64 or x86-64 host machine with Ubuntu 22.04, 64GB of free disk space and 32GB of RAM is minimum requirement to sync and build the platform software stack. 48GB of RAM is recommended though.

Git and Repo tool setup

The Neoverse RD software stack is available in multiple git repositories. In order to simplify downloading the software stack, repo tool can be used. This section explains the procedure to setup git and repo tool.

- Install Git by using the following command

```
sudo apt install git
```

- Git installation can be confirmed by checking the version

```
  git --version

This should return the git version in a format such as ``git version 2.7.4``
```

- Configure name and email address

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

- Install the repo tool by following these instructions.

This completes the setup of git and repo tool.

Platform Manifest Names

The repo tool uses a manifest file in order to download the source code. The manifest file lists the location of the various repositories and the branches in those repositories from which the code has to be downloaded. Each of the Neoverse RD platform has a unique manifest that is supplied to the repo tool to download the corresponding platform software. The following table lists the platform names and the corresponding manifest file names. Make a note of the manifest file name for the platform of your choice as that is required for the subsequent instructions.

Reference Platform	Manifest File Name
RD-Fremont	pinned-rdfremont.xml
RD-Fremont-Cfg1	pinned-rdfremontcfg1.xml
RD-Fremont-Cfg2	pinned-rdfremontcfg2.xml
RD-V2	pinned-rdv2.xml
RD-N2	pinned-rdn2.xml
RD-N2-Cfg1	pinned-rdn2cfg1.xml
RD-N2-Cfg2	pinned-rdn2cfg2.xml
RD-N2-Cfg3	pinned-rdn2cfg3.xml
RD-V1 (Single Chip)	pinned-rdv1.xml
RD-V1 (Quad Chip)	pinned-rdv1mc.xml
RD-N1-Edge (Single Chip)	pinned-rdn1edge.xml
RD-N1-Edge (Dual Chip)	pinned-rdn1edgex2.xml
SGI-575	pinned-sgi575.xml

Downloading the software stack

The manifest files, which contain the location of all the git repositories of RD platform software stack, are available here. This section explains the procedure to sync the software stack.

- Switch to a new empty folder

```
mkdir rd-infra
cd rd-infra
```

- To obtain the latest *stable* software stack, use the following commands listed below.

Note: In order to reduce the size of the commit history that is downloaded (and reduce the time taken to download the platform software stack), append "–depth=1" to the repo init command (without the quotes).

```
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>
repo sync -c -j $(nproc) --fetch-submodules --force-sync --no-clone-bundle
```

Note: The manifest file name for the required platform can be found in the *Platform Manifest Names* section. The RELEASE_TAG can be found in the `Release Tags` section of the corresponding platform's user guide or from the release notes, if obtained.

Note: 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* on resolving this issue.

Setting up the 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 contained based build is an another method in which container image is built from a container configuration file and has all the build dependencies satisfied and isolated from the host machine.

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.

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

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

Container Based

The supported container engine is docker and this setup is verified using Ubuntu 22.04 LTS as host OS.

The container image is designed to allow a user to have the sources directory in its 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 means that if you have already followed the section *Downloading the software stack* you can mount the folder inside the running container.

Install Container Engine

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

After installation is complete, refer to the post-installation steps on how to manage docker as non-root user. The container file, wrapper and utility scripts are located in this repository.

Note: Do not execute the wrapper script with root permissions.

The wrapper script *container.sh* defines the container file and image name by default and this can be changed either with flags *-f* and *-i* or by editing the file itself. To see all options available, change to the directory where you cloned the repository above and execute

```
cd container-scripts
./container.sh -h
```

Build Container Image

Note: Do not execute the wrapper script with root permissions.

To build the container image, execute

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

Run Container Image

Note: Do not execute the wrapper script with root permissions.

Mount the directory, in which the software stack has been downloaded, inside the container. This is achieved by using the flag *-v*. The mount point inside the container is `/home/$USER/workspace`

Note: The path needs to be an absolute path.

To run the container image, execute

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

You are now inside the container and the prompt shall look like

```
USER:HOSTNAME:~/workspace$
```

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

Setting up TAP interface (optional)

The platform FVP supports virtual ethernet interface to allow networking support to be usable for the software executed by the FVP. If support for networking is required, the host TAP interface has to be setup before the FVP is launched. To setup the TAP interface, execute the following commands on the host machine.

- Install libvirt and other packages

```
sudo apt-get install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils
```

- Ensure that the libvirtd service is active

```
sudo systemctl start libvirtd
```

- Use the ifconfig command and ensure that a virtual bridge interface named 'virbrX' (where X is a number 0,1,2,. . . .) is created. If there are no instances of virtual bridge available, use the following command to create it.

```
sudo brctl addbr virbr0
```

- Create a tap interface named 'tap0'

```
sudo ip tuntap add dev tap0 mode tap user $(whoami)
sudo ifconfig tap0 0.0.0.0 promisc up
sudo brctl addif virbr0 tap0
```

This completes the procedure to download the platform software stack, setup of the GCC toolchain binaries and installation of the other prerequisites. Refer the *troubleshooting guide* for solutions to known issues that might arise during use of the platform software stack.

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

2.14.4 MCP sideband channel

MCP sideband channel 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 adminstrators 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.

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 commmands. 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*

Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Building and running MCP sideband channel

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
```

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
```

(continues on next page)

(continued from previous page)

```
[    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
[    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 tranfer 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 trasaction 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
```

(continues on next page)

(continued from previous page)

```
[    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
[    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
```

(continues on next page)

(continued from previous page)

```
[    0.013482] [PLDM_FW]:                              11
[    0.013709] [PLDM_FW]:                              12
[    0.013782] [PLDM_FW]:                              13
[    0.013982] [PLDM_FW]:                              14
```

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 multip-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

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

2.14.5 Virtio-P9

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.

Overview of Virtio-P9 device

Few Arm reference design Fixed Virtual Platforms (FVPs) for infastructure 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.

Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Build the platform software

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

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 filsystem from the target platform's busybox prompt.

```
umount /mnt
```

This completes the validation of Virtio-P9 component on Arm infrastructure reference design platforms.

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

2.14.6 Troubleshooting Guide

Introduction

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 and execution stages. This page provides solutions for known issues that could affect the use of the platform software stack.

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
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
```

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
```

FVP closes abruptly

Tests such as distro installation take few hours to complete on Neoverse Reference Desgin 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" (without the quotes) parameter appended to the repo init command. The parameter "–depth=1" reduces the commit history that is downloaded and can reduce the failures in downloading linux repo.

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 *repo setup* page.

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
```

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

2.14.7 Guidelines to Vulnerability reporting

- 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.

Copyright (c) 2021-2024, Arm Limited. All rights reserved.

2.15 Fremont Documents

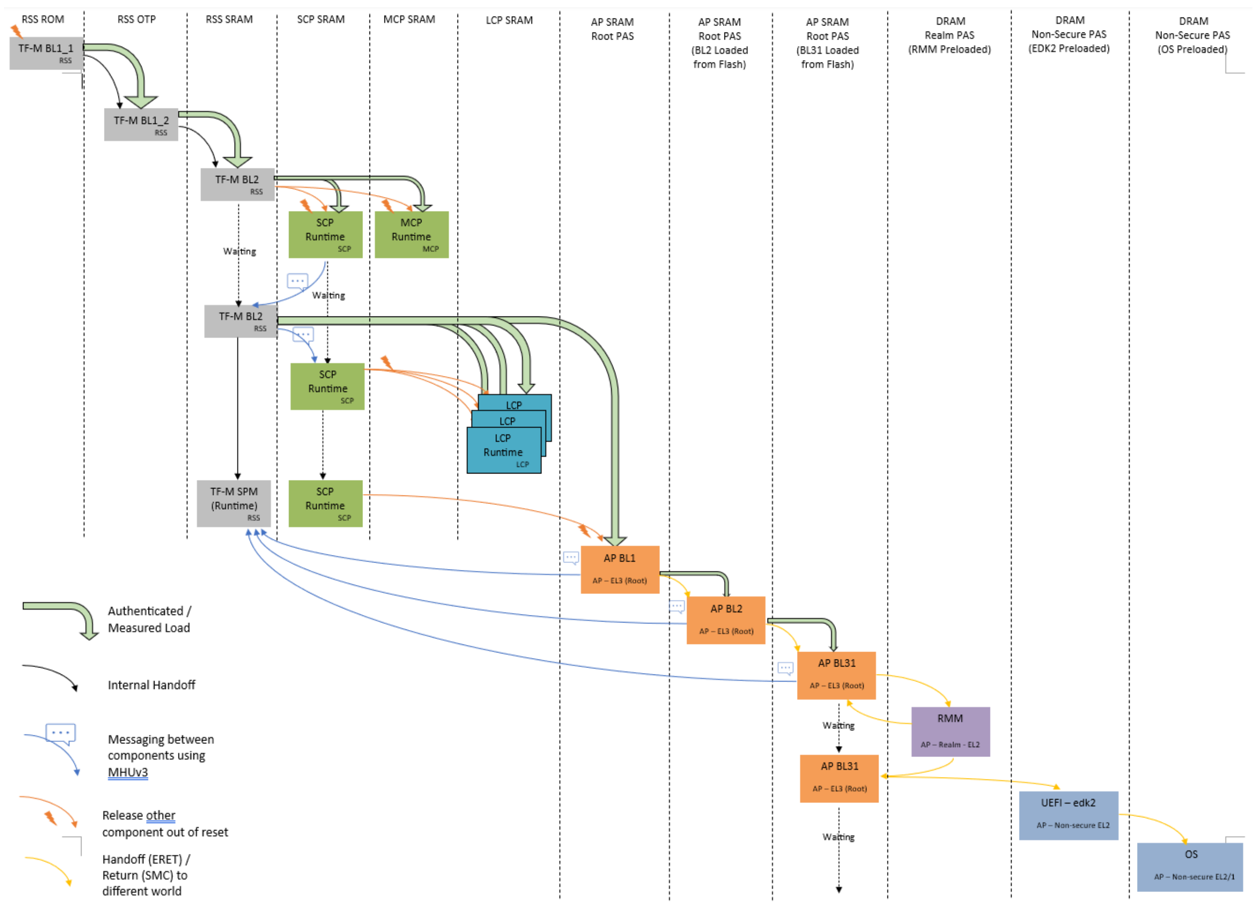
2.15.1 Boot Flow for RD-Fremont platform variants

Introduction

This page describes the overview of the software boot process on RD-Fremont platform variants.

Overview

A simplified boot flow diagram is shown below.



RSS

On platform reset, RSS is released out of reset and SCP and MCP are held in reset state. RSS begin executing the TF-M's BL1_1 from RSS ROM and provision the BL1_2 image into the One Time Programmable flash (OTP) and transfers the execution to the BL1_2 stage. More details on the provisioning can be found in the TF-M's RSS provisioning page. BL1_2 authenticates and loads the TF-M's BL2 (MCUBoot) stage which is responsible for authenticated loading of the next stage images as well as images of the other components. More details on BL2 can be found in *Image Loading Using MCUBoot* page. BL2 stage is also responsible for *setting up the SCP's ATU* and releasing SCP and MCP out of reset.

SCP

SCP is responsible for managing the system power and setting up of the interconnect. More details on the interconnect setup can be found in *CMN Cyprus Driver Module* page. SCP is also responsible for releasing the LCPs out of reset after RSS loads the LCP firmware into ITCMs. RSS has to wait for the SCP to turn on the SYSTOP power domain before loading the LCP images. To maintain this synchronization, SCP and RSS communicates messages through MHUv3. More details on this can be found in *SCP - RSS Communication* page.

LCP

LCP is responsible for managing per Application Processor's power. The boot sequence of LCP is summarized in *Local Control Processor* page.

AP

Application Processor (AP) is responsible for hosting the user's software stack. Trusted Firmware (TF-A) is executed at boot time to setup the root monitor, which executes at root EL3 mode. User's software stack includes a host hypervisor hosted at Non-Secure EL2 mode, a Realm Management Monitor (TF-RMM) deployed at Realm EL2 mode, guest virtual machine spawned in Realm/Non-secure EL1 and host/guest userspace applications running at Realm/Non-secure EL0.

In RD-Fremont platform, TF-M BL2 (MCUBoot) stage authenticates and loads TF-A BL1 image binary into AP SRAM. When SCP releases AP out of reset, AP starts with TF-A BL1 stage, following which TF-A BL1 authenticates and loads TF-A BL2 and passes the control to it. Furthermore TF-A BL2 follows same process to load the next stage TF-A BL31. TF-A BL31 is the runtime root monitor which sets up the TF-RMM interface and triggers the RMM initialization. TF-RMM communicates with TF-A BL31 on few occasions during its init. Once TF-RMM returns control back, TF-A BL31 hands off the control to Non-Secure BL33 image which is UEFI in current implementation which then take care of host OS/hypervisor boot. At each stage of authenticated loading of image, measurements of the loaded image is computed and extended to RSS, that is later used to produce delegated attestation key and token required by TF-RMM. More detailed info can be found here in *AP - RSS Attestation Service*.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.2 Runtime Security Subsystem

Introduction

The Runtime Security Subsystem processor (RSS) is tasked with being the secure root of trust for platforms it is included. The main tasks are securely loading firmware and provides secure services such as storing encryption keys safely and secure encryption/decryption without exposing the stored keys as part of CCA Hardware Enforced Security (CCA HES) (see *CCA Security Model*). In addition RSS provides attestation services and can store and return measurement data.

RSS firmware is a platform of the open source project Trusted Firmware-M (TF-M).

For further documentation see:
`<workspace>/tf-m/docs/`

and
`<workspace>/tf-m/docs/platform/arm/rss/`

Hardware Specification

RSS integrates Cortex-M55 processor core, two volatile memory banks, Memory Protection Controller (MPC), Non-Volatile ROM. In addition, RSS uses the following peripherals:

- Address Translation Unit (ATU) - Maps part of RSS address space to address space of other subsystems.
- Crypto Accelerator - A private accelerator to perform encryption and decryption operation without involvement of the processor.
- Key Management Unit (KMU) - Stores keys securely in a way they can only be accessed by Crypto Accelerator ensuring keys are secure even from RSS.
- One Time Programable memory (OTP) - Memory that allows data to only be written once.

Bootloader

The RSS Bootloader is split into 3 parts: BL1_1, BL1_2 and BL2 to provide a balance between security and ease of update.

BL1_1 is run from the ROM and has the main job of initializing the OTP, loading and running the provisioning bundles, and loading BL1_2 from the OTP into SRAM. It is kept relatively simple as it cannot be changed once the ROM is manufactured. For more details on provisioning see: RSS Provisioning.

BL1_2 is loaded from OTP to run from SRAM and has the main job of initializing the flash and loading BL2 from the flash into SRAM. This is easier to change as it is in OTP which can be changed at manufacturing without changing hardware like the ROM.

BL2 is loaded from flash to run from SRAM and has the main job of loading and verifying the firmware for all subsystems. BL2 will load the firmware for SCP, MCP, all LCPs, AP BL-1 and the RSS runtime firmware. The ATU is required to access the address space of the other processors. To access LCP, the SCP needs to have setup the CMN so this stage also initializes communication to coordinate with the other processors (see *SCP - RSS Communication*). This is also the point where the SCP ATU is setup (see *SCP Address Translation Unit (ATU) Configuration*).

BL2 is a configuration of MCUBoot which uses headers and trailers to validate the images. For more details see *Image Loading Using MCUBoot*.

Lifecycle

The chip lifecycle is split into 4 states.

1. Chip Manufacture (CM)
2. Device Manufacture (DM)
3. Secure Enabled (SE)
4. Return to Manufacture (RTM)

During CM and DM stages, secure data is loaded into the OTP and KMU from the provisioning bundles for use in the SE stage. SE is the main stage the chip is used in and the only stage that boots past BL1_1. RTM is only used for decommissioning the chip.

Secure Encryption

In RSS systems with a crypto accelerator and KMU secure keys for encryption are only stored in the KMU. This provides an extra level of security as once keys are stored and locked in the KMU they cannot be accessed by software. The only way to use keys in the KMU is through a secure channel between the KMU and crypto accelerator. This means even if the RSS runs compromised code the crypto keys will still be secure as they are never in memory or accessible to the processor during SE state.

Note: In the current implementation the KMU is not locked for debugging.

If a crypto accelerator software cryptography must be used which will require either the KMU is not locked or it is not used. If software cryptography is used the keys will also need to be in memory. This is considered less secure.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.3 Image Loading Using MCUBoot

MCUBoot is a secure boot library used by Trusted Firmware-M (TF-M).

TF-M loads SCP, MCP and LCP images using MCUBoot from flash into designated RAM regions for execution. This is MCUBoot's ram-load configuration. During image signing/creation additional information such as load address, header size, padding, signing key etc. that the bootloader expects can be facilitated using 'imgtool'. Each mcuboot image is accompanied with a header region and a trailer region that hold the booloader information. After each boot image load, MCUBoot records boot measurements into the shared data region. For more info, go to MCUBoot Documentation.

Flash memory is partitioned according to the flash map and each flash area is identified with an id (Refer `<workspace>/tf-m/platform/ext/target/arm/rss/rdfremont/bl2/flash_map_bl2.c`). Flash memory can include multiple image regions each of which contains two image slots, namely, primary and secondary. Before authentication and execution, the image is loaded into the RAM. Following represents a rough overview of SCP image loading:

```
                                 Flash
 FLASH_BASE_ADDRESS <--- +--------------------+
                         ~                    ~             Seperate Header region
                         ~        ~~~         ~             +--------------------+ --->␣
↪RSS_HEAD_PHYS_BASE /
                         ~                    ~             |                    |     ␣
↪HOST_SCP_HEAD_REGION_S
FLASH_AREA_6_OFFSET <--- +--------------------+             |     ~ unused ~     |      ␣
↪(Logical Address)
                         | SCP primary image  |             |                    |
                         | +----------------+-------------> +--------------------+ --->␣
↪HOST_SCP_IMAGE_BASE_S
                         | |    Header      | |             |   Image Header     |      ␣
↪(Logical Address)
                         | +----------------+-------+       +--------------------+
                         | |                | |     |
                         | |     code       | |     |
                         | +----------------+ |     |
                         | |                | |     |                RAM
                         | |      TLV       | |     +-----> +--------------------+ --->␣
↪HOST_SCP_PHYS_BASE /
                         | |       &        | |             |                    |     ␣
↪HOST_SCP_CODE_BASE_S
```

(continues on next page)

(continued from previous page)

```
                     | |    padding     | |           |       Code         |       ␣
↪(Logical Address)
                     | |                | |           |                    |
                     | |                | |           +--------------------+
                     | +----------------+ |           |                    |
                     +--------------------+           |                    |
                     ~                    ~           |        TLV         |
                     ~        ~~~         ~           |         &          |
                     ~                    ~           |      padding       |
                     +--------------------+           |                    |
                                                      |                    |
                                                      +--------------------+
                                                      ~                    ~
                                                      ~        ~~~         ~
                                                      ~                    ~
                                                      +--------------------+
```

To load an image, `boot_go_for_image_id()` is invoked by passing the respective image id. `boot_platform_pre_load()` and `boot_platform_post_load()` functions are invoked before and after loading images respectively to map image specific ATU regions. Before loading the image, MCUBoot checks for address overlaps. After loading the image, it is validated and boot measurements are recorded. MCUBoot loads the image including the image header. Components such as SCP, expect the code region rather than the image header at the start of the RAM. To ensure this, ATU region of RSS is mapped such that the logical address is linear from RSS perspective, but the physical addresses are remapped such that the code region of the image is loaded to the start of the RAM, and the image header to a separate memory region.

ATU region base address and size should be aligned to the ATU page size, i.e, 8KB (0x2000) for RD-Fremont platform. The ATU region is configured such that the header alone is loaded into a temporary region (with ATU minimum size of 0x2000) and rest of the image is loaded in the RAM region. Due to the memory constraints of LCP, this temporary region is made common for SCP, MCP and LCP, at the bottom of MCP RAM region. The header size is fixed at `BL2_HEADER_SIZE` (0x400) for the the host images. So, the load address has an offset of `0x2000 - 0x400 = 0x1C00` from `HOST_SCP_HEAD_REGION_S` to ensure that the image header ends up in the bottom on the temporary region.

SCP and MCP images are loaded into single address, whereas the LCP images needs to be loaded into multiple LCPs ITCM address. Currently, this is handled by remapping ATU regions iteratively to redirect the boot load into respective LCP ITCM (Refer `<workspace>/tf-m/platform/ext/target/arm/rss/rdfremont/bl2/boot_hal_bl2.c:boot_platform_post_load_lcp()`). Additionally, recording of LCP boot measurements into the shared data is skipped after each image loads. A single common boot measurement for LCP is recorded after each LCP image is loaded, i.e., at the end of `boot_platform_post_load_lcp()`, since the measurement for LCPs are identical.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.4 NI-Tower Network-on-Chip Interconnect

Introduction

NI-Tower Network-on-Chip Interconnect is a configurable system-level interconnect that is compliant with AMBA protocols. It provides various configuration options for upstream/downstream interfaces, xSNIs and xMNIs. For more details, see NI-Tower NCI TRM.

Overview

In addition, NI-Tower has two address map type; static and programmable address map (PSAM). Also, NI-Tower implements optional Access Protection Units (APU) that the software can use to implement memory protection and isolation from devices with different integrity levels.

RSS provides with drivers to program the PSAMs and APUs of the system control NI-Tower and APUs of the peripheral block NI-Tower. This enables having robust flexibility for modifying address map and memory protection based on the software implementation of that platform.

Discovery

Discovery is an algorithm that is used to determine the structure of the NI-Tower configuration. This helps RSS identify the address offsets for NI-Tower domains, components and sub-features without the knowledge of any configuration details.

Discovery flow

Following is the hierarchy of NI-Tower configuration nodes:

1. Global configuration node
2. Voltage domain
3. Power domain
4. Clock domain
5. Component
6. Subfeature

Component type includes all PMUs, xSNIs and xMNIs. Subfeatures include APU, PSAM etc.

Following the common header structure for all domain level nodes:

```
struct ni_tower_domain_cfg_hdr {
    __IM uint32_t node_type;
    __IM uint32_t child_node_info;
    __IM uint32_t x_pointers[];
};
```

Following the common header structure for all component nodes (except FMUs):

```
struct ni_tower_component_cfg_hdr {
    __IM  uint32_t node_type;
    const uint32_t reserved_0[4];
    __IM  uint32_t interface_id_0_3;
    const uint32_t reserved_1[58];
    __IM  uint32_t num_subfeatures;
    const uint32_t reserved_2;
    struct {
        __IM uint32_t type;
        __IM uint32_t pointer;
    } subfeature[];
};
```

For more details, check Section 15.2 of NI-Tower TRM.

Programmable Address map

A programmable address map provides flexibility to configure or change address regions and targets after RTL configuration and provides more options than a static address map.

For more details, check Section 11.4 of NI-Tower TRM.

Access Protection Unit

The APU supports freedom with mixed memory protection requirements and preserves the integrity of critical memory and peripherals.

For more details, check Section 9.1 of NI-Tower TRM.

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.15.5 Local Control Processor (LCP)

Introduction

The local control processor (LCP) is introduced in the latest Arm Neoverse Reference Design platforms in order to support scalable power architecture, which enables per application processor 'Dynamic Voltage Frequency Scaling' (DVFS) to increase the responsiveness of the system to changes in the workload running on that application processor. Supporting individual core DVFS efficiently in a system requires additional power control functionality. DVFS control under a separate LCP is done to avoid overloading the system control processor (SCP) for a system with higher core count.

Reference design platforms with LCP support a scalable power control solution that is enabled by having one an LCP for each application core (AP) in the system under the management of the System Control Processor (SCP).

LCP specifications

- Cortex-M55 CPU.
- 64KB instruction (ITCM) and 32KB data (DTCM) memory.

Functionality

- Per-core DVFS, Thermal management, Max Power Mitigation Mechanism (MPMM) and Power limit enforcement.
- Sensor data collection: LCP uses System Monitoring and Control Framework (SMCF) to collect data from thermal sensors and activity monitors and sends the data to SCP for further processing.

Power domain

LCP is in AON power domain and SYSTOP reset domain.

Boot Sequence

1. The runtime security subsystem (RSS) authenticates and loads the LCP RAM firmware to the LCP ITCM (instruction tightly coupled memory).

   References:

   a. RSS getting the LCP images base address and size from cluster utility (refer: `<workspace>/tf-m/platform/ext/target/arm/rss/rdfremont/bl2/flash_map_bl2.c: boot_get_image_exec_ram_info()`).

   b. RSS loading the LCP binary from cluster utility (refer: `<workspace>/tf-m/platform/ext/target/arm/rss/rdfremont/bl2/boot_hal_bl2.c: boot_platform_pre_load(), boot_platform_post_load()`).

2. LCP comes out of reset when SCP turns on the SYSTOP power domain, but the boot pauses in the CPUWAIT state.

3. SCP releases the LCP from CPUWAIT state by programming the CPUWAIT register in system control block (SCB) after the inteconnect setup is done. (refer: `<workspace>/scp/product/rdfremont/module/platform_system/src/mod_platform_system.c: release_lcp()`).

Peripherals

DVFS Handler

The DVFS (Dynamic Voltage and Frequency Scaling) Handler allows a software agent to abstract the details of programming the DVFS settings for components' clock frequency and voltage level. The interface allows software to program in set of registers a frequency and voltage level, making the hardware responsible for managing the lower level details of applying these settings to the required components. This removes the need for software to support SoC specific and complex programming sequences and reduces the overall software load.

References:

1. Module: `<workspace>/scp/module/dvfs_handler/src/mod_dvfs_handler.c`
2. Example configuration: `<workspace>/scp/product/rdfremont/lcp_ramfw/config_dvfs_handler.c`

Message Handling Unit

Message handling unit (MHU) v3 is supported in LCP to enable communication between LCP-AP and LCP-SCP.

References:

1. Module: `<workspace>/scp/module/mhu3/src/mod_mhu3.c`
2. Example configuration: `<workspace>/scp/product/rdfremont/lcp_ramfw/config_mhu3.c`

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.6 SCP Address Translation Unit (ATU) Configuration

Introduction

The ATU provides means to convert the 32-bit logical address from the sub-system memory map to the physical address residing in the system memory map with rich security attributes.

The SCP is treated as a secure processor in the RD-Fremont platform variants and hence cannot access Root or Realm physical address space (PAS). But during the platform boot, the SCP is required to generate transactions to root PAS for configuring the CMN via the ATU.

Overview

The SCP can access the AP memory map via Address Translation Window which must be configured in the ATU. The ATU driver module in the SCP firmware configures the ATU regions.

The SCP ATU is configured with the following translation regions:

Region	Logical Address Range	Physical Address Range	PAS
Cluster Utility	0x6000_0000 - 0x6FFF_FFFF	0x2_0000_0000 - 0x2_FFFF_FFFF	Root
AP Shared SRAM	0x7000_0000 - 0x77FF_FFFF	0x0000_0000 - 0x07FF_FFFF	Root
GPC SMMU	0x7800_0000 - 0x780F_FFFF	0x3_0000_0000 - 0x3_000F_FFFF	Root
AP Peripherals	0x7810_0000 - 0x784F_FFFF	0x2F00_0000 - 0x2F3F_FFFF	Secure
CMN CFGM	0xA000_0000 - 0xDFFF_FFFF	0x1_0000_0000 - 0x1_3FFF_FFFF	Root

This SCP ATU driver implementation can be found in the in the following file in the software stack: `<workspace>/scp/module/atu/`

Runtime Configuration of ATU

In addition to the statically mapped translation regions if an additional ATU region needs to be configured during the runtime, then the ATU API can be used by the modules.

Please refer `<workspace>/scp/module/atu/doc/atu.md`

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.15.7 SCP - RSS Communication

Introduction

This document gives an overview of the SCP to RSS communication stack.

```
+-----------------+        +-----------+        +-----------+
|                 |        |           |        |           |
|   SCP PLATFORM  |<------>| TRANSPORT |<------>|   MHUv3   |
|                 |        |           |        |           |
+-----------------+        +-----------+        +-----------+
```

Overview

The RD-Fremont platform SCP communicates with the RSS via MHU doorbell during the boot sequence.

In SCP firmware, the scp_platform module binds to the transport module. The transport HAL module is bound to the MHUv3 driver.

The SCP transport module configuration info can be found in the following file in the software stack: `<workspace>/scp/product/neoverse-rd/rdfremont/scp_ramfw/config_transport.c`

- The transport channel `SCP_PLATFORM_TRANSPORT` is used by the scp platform module during the initial boot to perform handshake with the RSS firmware via MHU doorbell events.

The SCP MHUv3 driver configuration info can found in the following file in the software stack: `<workspace>/scp/product/neoverse-rd/rdfremont/scp_ramfw/config_mhu3.c`

- In this config data, the element `SCP2RSS_S_MHU_DBCH` is used to communicate with the RSS.
- The SCP Platform transport channel uses the doorbell channel 1, flag 0 to perform handshake with the RSS firmware.

SCP-RSS handshake

The RSS firmware BL2 stage loads the MCP, SCP and LCP firmware images. The LCP firmware must be loaded to the LCP ITCM (Instruction - Tightly Coupled Memory) which is accessed via the cluster utility region in the AP memory space through the RSS ATU. But, in order to access the LCP ITCM, the RSS must wait for the `SYSTOP` power domain to be in ON state. So, the RSS firmware loads the SCP firmware image, releases the SCP and waits for the SCP to turn on the `SYSTOP` power domain.

After the SYSTOP is turned on and the platform is initialized, the SCP platform system module triggers the SCP to RSS MHUv3 doorbell and waits for a doorbell back from RSS which would indicate that the RSS has successfully loaded the LCP firmware images to the LCPs.

The RSS firmware on receiving the doorbell from SCP, loads the LCP firmware to the LCP ITCMs and triggers RSS to SCP MHUv3 doorbell. The SCP platform system module receives the doorbell event, configures the LCP UART and releases all the LCPs from reset.

The LCP initialization code in the SCP platform module can be found in the function `platform_setup_lcp()` in the following file in the software stack: `<workspace>/scp/product/neoverse-rd/rdfremont/module/scp_platform/src/platform_lcp.c`

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.15.8 CMN Cyprus Driver Module

Introduction

The CMN-Cyprus is a scalable configurable coherent interconnect that is designed to meet the Power, Performance, and Area (PPA) requirements for Coherent Mesh Network systems that are used in high-end networking and enterprise compute applications. This document gives an overview of the cmn cyprus driver module in the SCP firmware.

NOTE: The Super Home Node (HN-S) introduced in the CMN Cyprus is responsible for managing part of the local address space and managing local coherency for part of the remote address space. However, any descriptions related to the Fully coherent Home Node (HN-F) in this document also apply to the HN-S node.

Overview

The cmn cyprus driver module in the SCP firmware discovers the CMN interconnect present in the system and configures the CMN during the boot-time based on the address map info provided from the module configuration data.

The cmn cyprus module config data for RD-Fremont platform can be found here: `<workspace>/scp/product/neoverse-rd/rdfremont/scp_ramfw/config_cmn_cyprus.c`

The cmn cyprus driver module code can found in the following path: `<workspace>/scp/module/cmn_cyprus/`

NOTE: The CMN Cyprus configuration register space located in the AP memory map is accessible through Root access only. Hence, the Address Translation Unit in the RD-Fremont platform variants is configured in such a way that the SCP can access the CMN Configuration space with root permissions. Please refer to the *SCP ATU Configuration* document for more info on this.

HN-S Isolation

MXP Device Isolation feature in the CMN Cyprus allows HN-F or HN-S devices attached to a CAL2 device ports to be logically isolated from the corresponding MXP and mesh. If the isolation parameter is enabled in the mesh, then, by default, all the CAL2 connected HN-S devices are in isolated state.

Note: Isolated HN-S nodes must not accessed before disabling the isolation.

The MXP `mxp_device_port_disable` register of the corresponding MXP must be configured in order to bring the HN-S nodes out of isolation before accessing the HN-S nodes. The CMN Cyprus driver module config data has provision for passing a list of isolated HN-S node addresses and the coordinates. This info is used by the driver during the discovery to skip the isolated HN-S nodes(if any) and enable all the other MXP device ports before discovering the individual nodes connected to the cross-point.

Please refer `mxp_enable_device()` function in the `<workspace>/scp/module/cmn_cyprus/src/cmn_cyprus_mxp_reg.c` file for more info.

Sequence

The CMN Cyprus configuration space is located in the SYSTOP power domain which requires the SYSTOP power domain to be turned on and clock to be configured before the CMN CFGM space can be accessed. Hence, the CMN Cyprus driver starts initialization after the clock and the power domain modules.

The `cmn_cyprus_setup()` function is invoked during the start phase. This function initiates the CMN boot-time programming sequence.

Note: In Multichip platforms, the CMN Cyprus driver in each chip's SCP firmware programs the local CMN mesh, CCG blocks and enables the CML links as described in the following sections.

Discovery

1. During the module start phase, the `cmn_cyprus_setup()` function is invoked, which invokes `cmn_cyprus_discovery()` function to initiate the discovery.
2. The `cmn_cyprus_discovery()` function invokes the `discover_mesh_topology()` function which starts from the Rootnode base and traverses through the mesh to discover the interconnect topology and total number of each type of node present in the mesh.

- During the discovery, `disable_hns_isolation()` function is used to disable mxp device isolation for non-isolated nodes attached in the MXP before traversing the MXP. In addition to this, before reading the nodes info register, each node address is compared with the list of isolated HN-S nodes using the `is_node_base_isolated()` function and if a match is found, the corresponding child pointer of the cross-point is skipped.

3. After the discovery, the `cmn_cyprus_discovery()` allocates the space for the internal descriptors and pointers and invokes `cmn_cyprus_init_ctx()` function.
4. In the `cmn_cyprus_init_ctx()` function, the driver traverses through the mesh starting from the Rootnode base and each RN-SAM, HN-F node, CCG node info is stored in the module context data. This completes the CMN discovery sequence.

Please refer the following for more details: `<workspace>/scp/module/cmn_cyprus/src/cmn_cyprus_discovery_setup.c`

HN-F SAM Programming

After the discovery, `cmn_cyprus_setup()` invokes the function `cmn_cyprus_setup_hnf_sam()` to program the HN-F SAM.

1. This function checks the `hnf_sam_mode` in the `hnf_sam_config` and if Direct SN mapping mode is configured, then the function `program_hnf_sam_direct_mapping()` is invoked, which programs the SN-F node ID provided in the config data at the SN0 index.
2. If range based hashed SN mapping mode is enabled, then the function `program_hnf_sam_range_based_hashing()` is invoked where the target SN node IDs are configured along with the top address bit positions and finally the SN mode is enabled. At the moment, only 3-SN mode is supported.
3. Next, `program_syscache_sub_regions()` is invoked which iterates through each region in the memory map and for each system cache group sub-region i.e regions with the type configured as `MOD_CMN_CYPRUS_MEM_REGION_TYPE_SYSCACHE_SUB` (region serviced by dedicated SBSX node for non-hashed access) the address range is configured in the HN-F SAM in the `configure_non_hashed_region()` function.
4. Finally, the HN-F node's PPU(Power Policy unit) register is configured.

Please refer `<workspace>/scp/module/cmn_cyprus/src/cmn_cyprus_hnsam_setup.c` file for more details.

LCN SAM Programming

For multichip platforms that enable Local Coherency Node feature, the function `cmn_cyprus_setup_lcn_sam()` is invoked to program the remote hashed regions in the LCN SAM. The address range, target type, CAL mode and CPA are programmed.

RNSAM Programming

Following the HN-F SAM programming, `cmn_cyprus_setup()` function invokes `cmn_cyprus_setup_rnsam()` to program the RN SAM.

1. `stall_rnsam_requests()` function is invoked to stall the RNSAM requests initially.
2. For each region provided in the config mmap table, `program_rnsam_region()` is invoked, which checks the memory region type and invokes the function `program_io_region()` if the region is an I/O region or `program_scg_region()` function if the region is a System cache backed region.

Non-Hashed regions

Non-hashed memory regions are used when a memory partition has to be assigned to an individual target node. I/O space is the intended target of these regions.

1. The region type is `MOD_CMN_CYPRUS_MEM_REGION_TYPE_IO` and the function `rnsam_configure_non_hashed_region()` is invoked to configure the address range and the target node type.
2. The address range and the target node type is configured in the following RNSAM registers when range comparison mode is enabled.
   - `non_hash_mem_region_reg`
   - `non_hash_mem_region_cfg2_reg`
3. After that, the target node id is configured in the register `non_hash_tgt_nodeid` in `rnsam_set_non_hashed_region_target()` function and the region is marked as valid for comparision in the function `rnsam_set_non_hash_region_valid()`.

Hashed regions

Hashed memory regions are used when a memory partition is distributed across many target nodes. These regions are typically used to support DRAM space to be hashed across multiple HN-F nodes.

1. The region type is `MOD_CMN_CYPRUS_MEM_REGION_TYPE_SYSCACHE` and the function `rnsam_configure_hashed_region()` is invoked to configure the address range and the target node type.
2. The address range and the target node type is configured in the following RNSAM registers when range comparision mode is enabled:
   - `sys_cache_grp_region`
   - `hashed_tgt_grp_cfg2_region`
3. Optionally, the secondary address range is also configured in the following RNSAM registers when range comparision mode is enabled:
   - `sys_cache_grp_secondary_region`
   - `hashed_tgt_grp_secondary_cfg2_region`
4. Next, the CAL2 (Component Aggregation Layer) mode is configured if specified in the config data.
5. Then, if hierarchical hashing mode is specified in the config data, the function `configure_scg_hier_hashing()` is invoked, which configures the number of clusters in the first level hierarchy, number of HN-S nodes per cluster, number of address bits to be removed at second level and enables the hierarchical hashing mode using RNSAM utility functions.
   - The following registers are configured:
     - `hashed_target_grp_hash_cntl_reg`
     - `sys_cache_grp_sn_attr`
     - `sys_cache_grp_sn_sam_cfg`
6. After this, the hashed region is marked as valid for comparision.
7. Once the hashed memory region is programmed, the function `configure_scg_target_nodes()` programs the target HN-S node IDs that fall within the SCG region boundary specified in the config data.

Note: An SCG is a group of HN-Fs that share a contiguous address region. However, the addresses that are covered by each HN-F in an SCG are mutually exclusive. An HN-F belonging to an SCG is selected as the target based on a hash function.

- Typically, these HN-F nodes are bound by an arbitrary rectangle in the mesh and the module config data specifies the start and the end HN-F node positions of the SCG/HTG. If the node falls within the SCG, then the node id is programmed in the following RN-SAM registers:
  - `sys_cache_grp_hn_nodeid_reg`

8. Then, `sys_cache_group_hn_count` register is programmed with the number of HN-Ss in the hashed target group.
9. For multichip platforms, the remote chip memory regions and the CML port aggregation mode is also configured by the function `program_rnsam_remote_regions()`.
10. Finally, the RNSAM is enabled by configuring the `rnsam_status` register, where the requests are unstalled and default target id selection is disabled in the `rnsam_unstall()` function.

Please refer `<workspace>/scp/module/cmn_cyprus/src/cmn_cyprus_rnsam_setup.c` for more details.

CML Programming

For multichip platforms, the function `cmn_cyprus_setup()` invokes `cmn_cyprus_setup_cml()` to program the CML links and enable multichip communication.

The multichip connection is setup in the following sequence:

1. The remote memory regions are programmed in the RA SAM. The function `program_ra_sam()` is invoked to configure the remote address range, target HAID and set the RA SAM region as valid.
2. Assign LinkIDs to remote agents. `program_agentid_to_linkid_lut()` function programs the LinkID in the AgentID-to-LinkID register. The default value points to Link 0. Hence, the driver skips the configuration and sets the linkid programming as valid.
3. Program the Home Agent ID (HAID) in the CCG HA node using the function `ccg_ha_configure_haid()` function.
4. Assign expanded RAIDs to local request agents. The function `program_ccg_ra_ldid_to_raid_lut()` programs unique RAID values for the local RN-Fs, RN-Is, RN-Ds and optionally LCN in the respective LUT registers.
5. Assign LDIDs to remote caching agents. The function `program_ccg_ha_raid_to_ldid_lut` programs LDID values for each remote caching agent in the CCG HA RAID-to-LDID LUT registers.
6. Program CCG HA node IDs at the LDID index for remote caching agents in the HN-S nodes using the function `program_hns_ldid_to_rn_nodeid()`. The request nodes are set as remote and based on the config data, CPA is also configured.
7. Then, if the SMP mode is specified in the config data, the functions `ccg_ra_set_cml_link_ctl()` and `ccg_ha_set_cml_link_ctl()` are invoked to configure the SMP mode in CCG RA and CCG HA registers respectively.
8. If direct connect mode (ULL-to-ULL) mode is required, the function `enable_ccla_direct_connect_mode()` configures the CML direct connect mode.
9. The CML link is enabled, the link status is then verified and the link is brought up using the function `setup_cml_link()`.
10. Then, `cml_exchange_protocol_credit()` function is used to exchange the protocol credits between the CML links.

11. Finally the `cml_enter_coherency_domain()` function is used to enable the CML links to enter snoop coherency domain and DVM domain.

Copyright(c) 2023-2024, Arm Limited.All rights reserved.

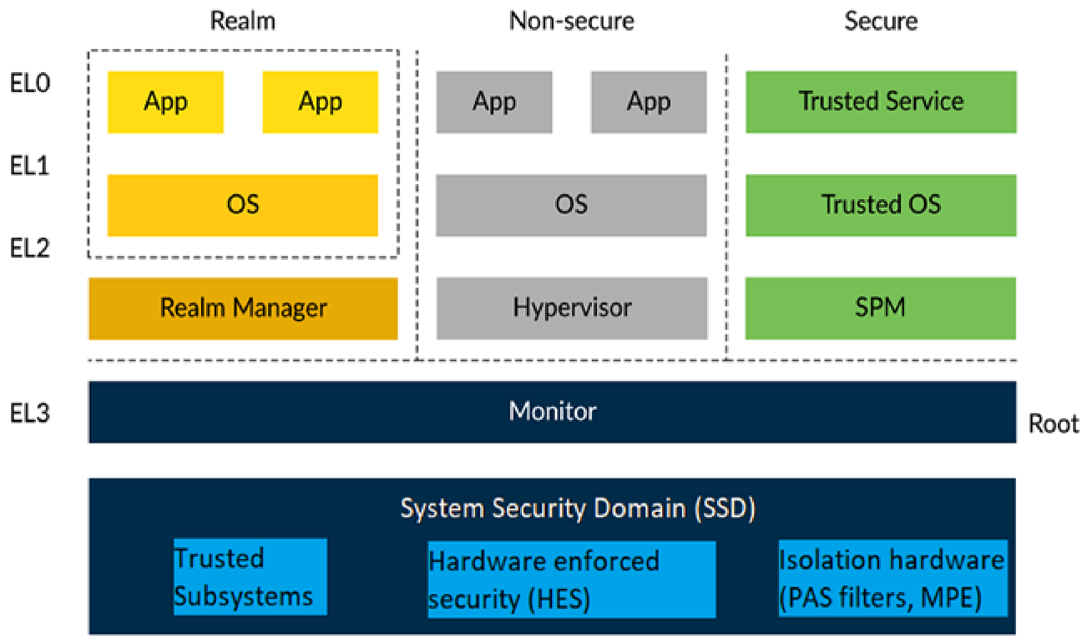
2.15.9 Realm Management Extension (RME)

Introduction to Confidential Compute architecture

In computing, data exists in three states: in transit, at rest, and in use. Data traversing the network is "in transit," data in storage is "at rest," and data being processed is "in use." Today, data is often encrypted 'at-rest' in storage and 'in-transit' across network, but not while 'in-use' in memory and process engine. Before the data can be processed by an application, it must be unencrypted in memory. This leaves the data vulnerable during processing when attackers may target data in memory or during processing by processing engine. Confidential Computing is a technology to protect the 'data-in-process' by processing the data in a hardware based protected CPU enclave or trusted execution environments (TEEs). The TEE is secured with encryption keys that prevent malicious foreign execution from attacking and breaching confidential data use.

Arm Confidential Compute architecture (CCA) allows to deploy isolated hardware trusted execution environment called as Realms. Realms allow lower-privileged software, such as an application or a Virtual Machine to protect its content and execution from attacks by higher-privileged software, such as an OS or a hypervisor. The hypervisor however, manages system resources.

The diagram below gives the security model for Arm CCA. For more details refer to the Arm CCA security model.



RME security model

RME architecture

Realm Management Extension or RME is an extension to Arm v9 arcitecture that defines the set of hardware features and properties that are required to comply with the Arm CCA architecture. As shown in above diagram the RME extends the Arm security model to four states now:

a) Secure state

b) Non-secure state

c) Realm state

d) Root state

SCR_EL3.{NSE,NS} controls PE security state.

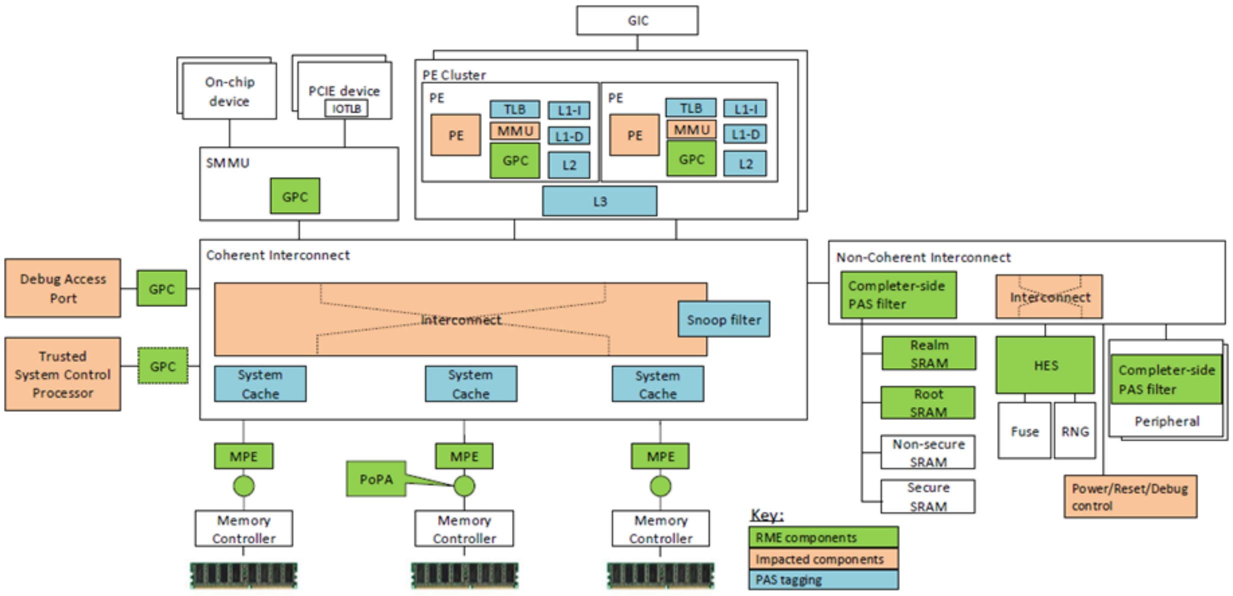
SCR_EL3.{NSE,NS}	Security state
0b00	Secure
0b01	Non-secure
0b10	NA
0b11	Realm

There is no encoding for root state. While in Exception level 3, the current security state is always root, regardless of the value of SCR_EL3.{NSE,NS}.

RME provides hardware-based isolation. In that:

- Execution in the Realm security state cannot be observed or modified by an agent associated with either the Non-secure or the Secure security state.
- Execution in the Secure security state cannot be observed or modified by an agent associated with either the Non-secure or the Realm security state.
- Execution in the Root security state cannot be observed or modified by an agent associated with any other security state.
- Memory assigned to the Realm security state cannot be read or modified by an agent associated with either the Non-secure or the Secure security state.
- Memory assigned to the Secure security state cannot be read or modified by an agent associated with either the Non-secure or the Realm security state.
- Memory assigned to the Root security state cannot be read or modified by an agent associated with any other security state.

Below is an example system architecture for a RME enabled platform.



RME system architecture

As seen above the RME enabled platform may include following system components:

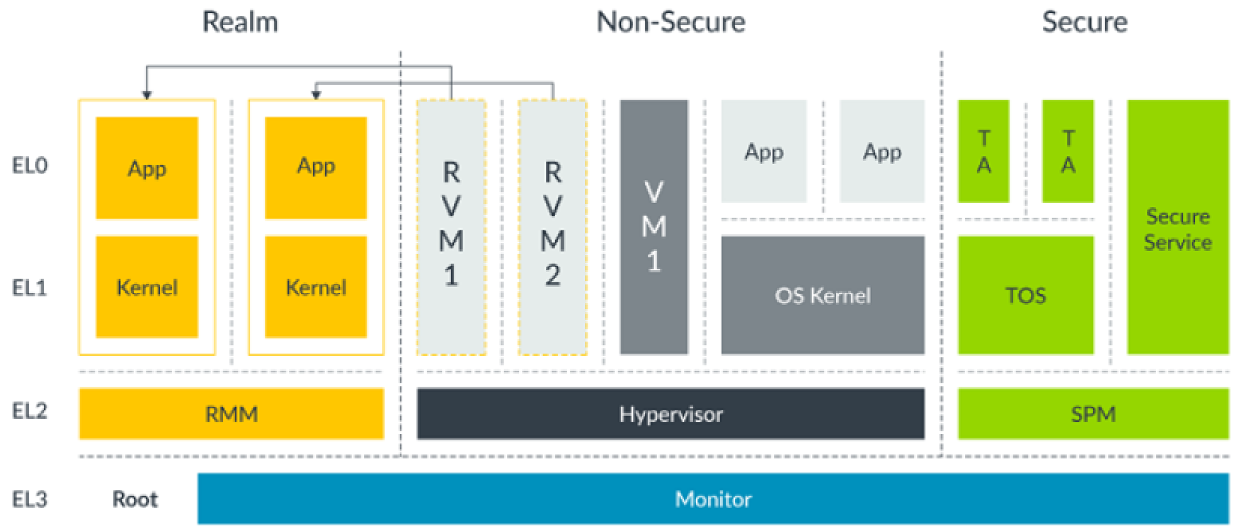
a) Requester side filters (GPC - placed after MMUs)

b) Completer side filters (embedded in interconnects/devices)

c) Memory Protection Engine (MPE – placed before memory controller)

RME software architecture

Below diagram show the participating software components in an RME software architecture.



RME software components

The platforms implementing these software components may follow a boot flow as mentioned in the *Boot flow*.

The software components that add support for Arm RME architecture are -

a) Trusted firmware for M class (TF-M)

b) Trusted firmware for A class (TF-A)

c) Realm Management Monitor (TF-RMM)

d) Linux kernel

e) Linux KVM tool (lkvm)

f) KVM unit tests

Platforms implementing RME architecture must also includes a CCA HES (hardware enforced security) module that serves as the root of trust in the CCA chain of trust. Runtime security subsystem (RSS) provides this support.

Validating RME support on RD-Fremont

In order to validate RME architecture support on RD-Fremont platform following test cases should succeed.

a) Parallelly booting virtual machine in the Realm and the non-secure security state,

b) Running Realm kvm-unit-tests that test a number of Linux KVM unit tests fo Realm security state.

Follow the *Virtualization* guide for more details on virtualization support on Neoverse reference design platform.

Build the RD-Fremont platform software stack for buildroot root filesystem by following the *Buildroot guide*.

Now boot the RD-Fremont fastmodel with the software stack as given in the *Buildroot guide* with additional model parameters to enable support for Scalable Vector Extensions (SVE). For example, to launch the RD-Fremont model to buildroot prompt with SVE:

```
./boot-buildroot.sh -p rdfremont -a  "--plugin /absolute/path/to/
↪ScalableVectorExtension.so -C SVE.ScalableVectorExtension.veclen=16 -C SVE.
↪ScalableVectorExtension.enable_at_reset=0" -n [true|false]
```

After boot mount the disk-image that contains Linux kernel for guest, lkvm tool and kvm-unit-tests built with RME support.

a) Running a virtual machine in Realm security state

- After booting the platform the disk-image partition can be mounted as:

```
# mount /dev/vda2 /mnt
```

- This partition should contain the following files and directories:

```
# cd /mnt
# ls
Image                    kvmtool                  ramdisk-buildroot.img
kvm-ut                   lost+found
```

- Now launch a virtual machine in Realm security state using lkvm with Linux and buildroot ramdisk images:

```
# screen -md -S "<screen_name>" /mnt/kvmtool/lkvm run --realm -c 2 -m 512 -
↪k /mnt/Image -i /mnt/ramdisk-buildroot.img --console serial -p earlycon
```

- Considering the <screen_name> in above command is “realm” verify that the screen session is running:

```
# screen -ls
There are screens on:
   198.realm       (Detached)
```

- Jump to the screen session to see the virtual machine boot in Realm security state:

```
# screen -r realm    // <screen_name: realm>
```

- Switch back the console to host using screen command shortcut 'Ctrl-a d'. Now launch a virtual machine in non-secure security state as well:

```
# screen -md -S nsvm /mnt/kvmtool/lkvm run -c 2 -m 512 -k /mnt/Image -i /
↪mnt/ramdisk-buildroot.img --console serial -p earlycon
```

- Verify that both screen sessions are running:

```
# screen -ls
There are screens on:
   198.realm       (Detached)
   214.nsvm        (Detached)
```

This completes the launch of virtual machine in Realm security state.

b) Running Realm kvm-unit-tests

Separate set of kvm-unit-test are developed to test various RME architectural features. Some of these include tests for Realm-RSI (Realm Service Interface), PSCI support in realm security state, GIC to validate interrupt support in the Realm, and Realm attestation to check support for remote attestation for realm payloads. As mentioned above build and boot the RD-Fremont platform with build- root filesystem.

- After booting the platform the disk-image partition can be mounted as:

```
# mount /dev/vda2 /mnt
```

- This partition should contain the following files and directories:

```
# cd /mnt
# ls
Image                   kvmtool                 ramdisk-buildroot.img
kvm-ut                  lost+found
```

- Now run the realm kvm-unit-tests from kvm-ut/ directory.

```
# cd kvm-ut/arm/
# export LKVM=/mnt/kvmtool/lkvm
# ./run-realm-tests
```

- This will launch a number of KVM unit tests that are run in Realm security state. An example output looks like below:

```
# lkvm run -k ./selftest.flat -m 16 -c 2 --name guest-200
PASS: selftest: setup: smp: number of CPUs matches expectation
INFO: selftest: setup: smp: found 2 CPUs
PASS: selftest: setup: mem: memory size matches expectation
```

(continues on next page)

(continued from previous page)

```
INFO: selftest: setup: mem: found 16 MB
SUMMARY: 2 tests

# KVM session ended normally.
# lkvm run -k ./selftest.flat -m 16 -c 1 --name guest-222
PASS: selftest: vectors-kernel: und
PASS: selftest: vectors-kernel: svc
SKIP: selftest: vectors-kernel: pabt test not supported in a realm
SUMMARY: 3 tests, 1 skipped

# KVM session ended normally.
# lkvm run -k ./selftest.flat -m 16 -c 1 --name guest-244
PASS: selftest: vectors-user: und
PASS: selftest: vectors-user: svc
SUMMARY: 2 tests

# KVM session ended normally.
# lkvm run -k ./selftest.flat -m 32 -c 4 --name guest-266
INFO: selftest: smp: PSCI version: 1.1
INFO: selftest: smp: PSCI method: smc
INFO: selftest: smp: CPU  1: MPIDR=0080000001
INFO: selftest: smp: CPU  2: MPIDR=0080000002
INFO: selftest: smp: CPU  0: MPIDR=0080000000
INFO: selftest: smp: CPU  3: MPIDR=0080000003
PASS: selftest: smp: MPIDR test on all CPUs
INFO: selftest: smp: 4 CPUs reported back
SUMMARY: 1 tests
```

This completes the Realm kvm-unit-tests run.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.10 AP - RSS Attestation Service

Introduction

This document gives an overview of handling of all the secure service requests that originates from TF-A running on AP and lands in TF-M SPM running on RSS. Arm CCA platforms has certain security requirements. One of the requirement is to provide with a platform attestation token whenever any user code executing in Realm demands for it. This attestation token is used by the Realm user to verify the trustworthiness and credibility of the platform. In simpler terms this attestation token is like a compiled signed document that has all the details about the hardware lifecycle state, SHA hash of all the trusted binaries which are loaded on the hardware and other claims about the legitimacy of the platform. To learn more about this please refer to Arm CCA.

On RME enabled RD platforms TF-RMM, TF-A and TF-M are involved in creation and transportation of the attestation token to Realm user. This document only focuses on providing a surface level understanding of the complicated boot time communication between TF-RMM, TF-A and TF-M for exchange of attestation token related data. The actual process of key and token generation in TF-M and transportation of the same from RMM to real user is out of the scope of this document.

Overview

There are two communication interfaces used to obtain realm private key and attestation token:

1. Interface between TF-A running on AP and TF-M SPM running on RSS.
2. Interface between TF-A and TF-RMM both running on AP.

TF-M has a boot time and runtime phase. TF-M runtime phase has all the secured services running as a separate thread in a multithread model. All the request always begins from AP side and RSS responds to those request by handling it through relevant secure services. TF-A also has boot time and runtime stages. In current situation all TF-RMM requests are catered by EL3 runtime TF-A stage which in certain conditions forwards the request to RSS for it to handle.

The overall attestation token is a composition of sub-tokens, where each sub-token is generated by different entitiy in the system. And each of the sub-tokens are cryptographically bound to each other. On RME enabled RD platforms from TF-M, to TF-A to all the way till TF-RMM, there are two sub-tokens. One of the token is called Initial Attestation Token (IAT) which is provided by RSS that includes measurements of all the updatable images running on AP, RSS, MSCP and LCP, hardware lifecyle state and other claims. The other sub-token is generated by TF-RMM which falls out of the scope of this document. Together they are called realm attestation token. Here measurements are nothing but SHA hash calculation of trusted loaded binary images which is encapsulated along with other metadata info to form a measurment. To learn more about attestation token and measurments, please refer to Arm CCA Security Model.

Each sub-token is needed to be signed by a private key. The IAT is the first token which is signed by a key provisioned in RSS Key Management Unit (KMU). TF-RMM requests for a different private key called as realm private key which is derived by RSS and provided back to TF-RMM. To learn more about the cryptographic binding of two sub-tokens and the maintained hierarchy, please refer to Delegated Attestation Service Integration Guide.

TF-M and TF-A Interface

AP and RSS communicates over a secure gateway which is implemented in hardware called as Message handling Unit (MHU). On RME enabled RD platforms we have four MHU units connecting AP and RSS. Each unit's accessibility is restrictive to the security state in which the processing element (PE) is executing. These security states align to the four states mentioned in ARMv9 RME which are Non-secure, Secure, Realm and Root. Since TF-A BL31 is the root monitor in RME enabled RD platforms, while AP is executing TF-A code, it is always at Root EL3 mode. Similarly for the secure transmission and reception of data to and from RSS, Root MHU is used. This pathway is used to implement TF-M and TF-A software interface.

There are three specific services related to attestation that TF-M provides:

1. Record and extend measurements
2. Derive and provide a delegated attestation key
3. Provide the delegated attestation token

TF-A loads in stages. Staring with TF-A BL1 stage which is loaded by TF-M MCUboot. TF-M calculates and stores the measurement of TF-A BL1 image once it is loaded into AP SRAM. When AP boots it starts with TF-A BL1 which authenticates and loads TF-A BL2 image into AP SRAM and similarly when BL2 starts it authenticates and loads TF-A BL31 into AP SRAM. At each stage whenever there is a authenticated loading of any image, measurment of that image is also recorded. And this recorded measurement is provided to RSS through TF-M & TF-A MHU interface. This is called extending of measurment to RSS. This is a necessary step for TF-M to store all the measurements which is later used to derive delegated attestion key and compute delegated attestation token. The recording of the measurements are not just limited to TF-A BL images but also include FW configs which are Device tree blobs (DTB).

When BL31 comes up, there are no more images to load but it serves as a intermediary between TF-RMM requests and TF-M services. The request for delegated attestation key and token originally comes from TF-RMM. More about this process is explained in next section.

All the service request that arrives at TF-M from TF-A are initially picked up by a particular sevice named as `ns_agent_mailbox`. Repsonsibility of this service is to fetch message from MHU and pass it to relevant partitions like `measured_boot` & `delegated_attestation`. And once respective partitions have produced a response, it is gathered again by `ns_aganet_mailbox` and sent over MHU to TF-A. More detail on this can be read from SPE - NSPE communication.

TF-A and TF-RMM Interface

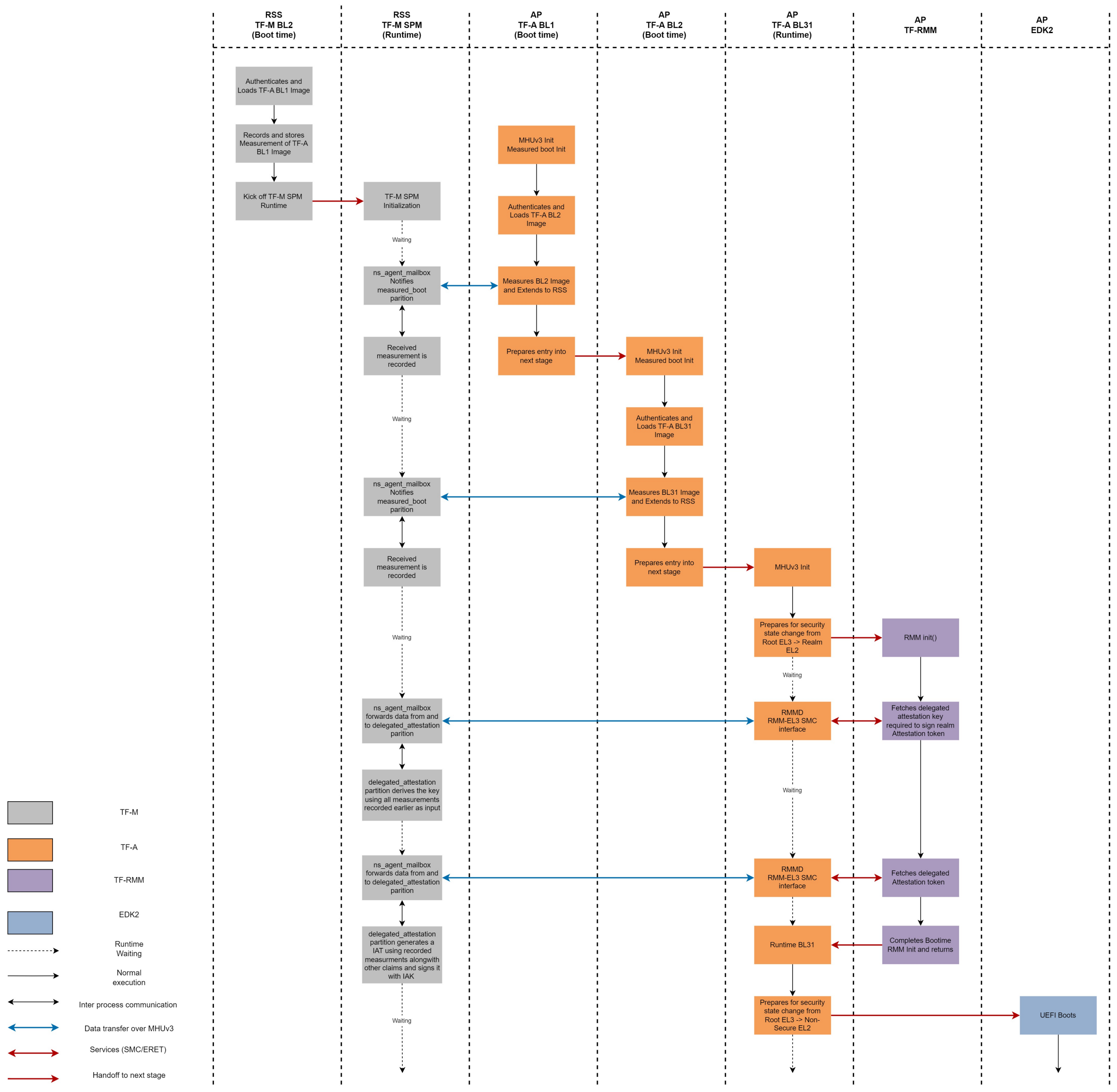
TF-A BL31 is the runtime stage which interfaces to TF-RMM. There are two parts in this interface: the boot interface and the runtime interface. To learn more about this interface refer to RMM-EL3 Communication Interface document. There are many runtime services which are requested by RMM over this interface to TF-A BL31. Two important services among these that needs special attention are `RMM_ATTEST_GET_REALM_KEY` and `RMM_ATTEST_GET_PLAT_TOKEN`.

As the name suggests, with `RMM_ATTEST_GET_REALM_KEY` TF-RMM request for a realm private key from TF-A BL31 and then TF-A BL31 request for delegated attestation key from TF-M SPM runtime. In response TF-M SPM runtime invokes `delegated_attestation` service which dervies the key and returns back the key to TF-A BL31 which is finally returned back to TF-RMM as a realm private key. During the derivation of private key, it uses CRYPTO service and `measured_boot` service to fetch all stored recorded measurments in RSS as these measurments are fed as input to the key derivation process.

As a next step TF-RMM requests for the IAT sub-token from TF-A BL31 using the service `RMM_ATTEST_GET_PLAT_TOKEN`. TF-A BL31 again forwards this as a deleagated attestation token request to TF-M SPM. And TF-M SPM invokes same service i.e. `delegated_attestation`. Internally it fetches the IAT from the `initial_attestation` service. This particular service has the duty to compile all the recorded measurments, hardware lifecycle state and other claims into a token and sign it. Once the IAT is produced, it is provided back to TF-A BL31 as a repsonse to request which is retuned back to TF-RMM.

Overall Flow

A simplified flow diagram is shown below.


RSS
TF-M BL2
(Boot time)
RSS
TF-M SPM
(Runtime)
AP
TF-A BL1
(Boot time)
AP
TF-A BL2
(Boot time)
AP
TF-A BL31
(Runtime)
AP
TF-RMM
AP
EDK2
Authenticates and Loads TF-A BL1 Image
Records and stores Measurement of TF-A BL1 Image
Kick off TF-M SPM Runtime
TF-M SPM Initialization
MHUv3 Init Measured boot Init
Authenticates and Loads TF-A BL2 Image
Waiting
ns_agent_mailbox Notifies measured_boot parition
Measures BL2 Image and Extends to RSS
Received measurement is recorded
Prepares entry into next stage
MHUv3 Init Measured boot Init
Waiting
Authenticates and Loads TF-A BL31 Image
ns_agent_mailbox Notifies measured_boot parition
Measures BL31 Image and Extends to RSS
Received measurement is recorded
Prepares entry into next stage
MHUv3 Init
Waiting
Prepares for security state change from Root EL3 -> Realm EL2
RMM init()
Waiting
ns_agent_mailbox forwards data from and to delegated_attestation parition
RMMD RMM-EL3 SMC interface
Fetches delegated attestation key required to sign realm Attestation token
delegated_attestation partition derives the key using all measurements recorded earlier as input
Waiting
ns_agent_mailbox forwards data from and to delegated_attestation parition
RMMD RMM-EL3 SMC interface
Fetches delegated Attestation token
delegated_attestation partition generates a IAT using recorded measurments alongwith other claims and signs it with IAK
Runtime BL31
Completes Bootime RMM Init and returns
Waiting
Prepares for security state change from Root EL3 -> Non-Secure EL2
UEFI Boots
TF-M
TF-A
TF-RMM
EDK2
Runtime Waiting
Normal execution
Inter process communication
Data transfer over MHUv3
Services (SMC/ERET)
Handoff to next stage

The complete sequence of measured boot and generation of platform attestation token is divided into these steps:

1. At the beginning TF-M BL2 (MCUBoot) loads the TF-A BL1 into AP SRAM and stores its measurments internally. At this moment there is no involvement from AP as it hasn't booted yet, so there is no requirment to use TF-M(RSS) and TF-A(AP) interface.

2. SCP bring AP out of reset to start executing the TF-A BL1. TF-A BL1 goes ahead and initializes other componenets and in the end authenticates and loads TF-A BL2 into AP SRAM and right after that it records the measurment of TF-A BL2 and extends it to RSS over TF-M & TF-A interface.

3. TF-A BL1 hands off the control to TF-A BL2. Which carries out many other operations before authenticating and loading BL31. And just like before it extends the recorded measurment of TF-A BL31 using same TF-M & TF-A interface.

4. Once TF-A BL2 is done with its functions, the control is passed to TF-A BL31 which sets up the RMM-EL3 interface to service requests arriving from TF-RMM and then invokes TF-RMM to perform its initialization.

5. During early init of TF-RMM, it requests for realm private key on TF-A & TF-RMM interface. Which is translated as a request for delegated attestation key by TF-A BL31 before being forwarded to TF-M on TF-M & TF-A interface. TF-M responds back with the key which is forwarded to TF-RMM by TF-A BL31.
9. On similar terms TF-RMM requests for final platform attestation token which is translated as a request for delegated attestation token by TF-A BL31 and forwarded to TF-M. TF-M uses internal secure services to produce a attestation token which is fetched by TF-A BL31 returned back to TF-RMM.
7. This completes the process of obtaining of the platform attestation token at boot time. Later during realm runtime, whenever a request for token arrives at TF-RMM from Realm user, it adds additional sub-token on top of platform attestation token to create a combined attestation token.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.11 Chain of Trust (CoT) for Confidential Compute Arcitecture (CCA)

Introduction

RD-Fremont platform has the support for trusted board boot (TBB), which prevents malicious firmware from running on the platform by authenticating all firmware images up to and including the BL33 bootloader (UEFI firmware).

The TBB Chain of Trust (CoT) starts with a set of implicitly trusted components. On the Arm development platforms, these components are:

- An SHA-256 hash of the Root of Trust Public Key (ROTPK). It is stored in the trusted root-key storage registers. In the Fremont reference design, the ROTPK and hash can be found in plat/arm/board/common/rotpk/ folder of TF-A and is intended for development purposes. This SHA256 hash of the ROTPK is embedded into the BL1 and BL2 images.
- The BL1 image: The APUs in RD-Fremont platforms are designed to prevent any unauthorized tampering or modification of the BL1 image.

The remaining components in the CoT are certificates and boot loader images. The certificates follow the X.509 v3 standard. It supports adding custom extensions to the certificates, which are used to store essential information to establish the CoT.

In the TBB CoT all certificates are self-signed. There is no need for a Certificate Authority (CA) because the CoT is not established by verifying the validity of a certificate's issuer but by the content of the certificate extensions.

The certificates are categorised as “Key” and “Content” certificates. Key certificates are used to verify public keys which have been used to sign content certificates. Content certificates are used to store the hash of a boot loader image. An image can be authenticated by calculating its hash and matching it with the hash extracted from the content certificate. RD-Fremont platform uses SHA256 hash algorithm.

Required keys

The keys used to establish the CCA vased CoT are:

Root of trust key

The private part of this key is used to sign CCA content certificate. CCA content certificate holds the hashes of BL2, BL31 and RMM images.

Platform root of trust key

The private part is used to sign platform key certificate. The platform key certificate holds the hash of non-trusted firmware content certificate, and the non-trusted firmware content certificate holds the hash of UEFI (BL33) binary.

Core secure world key

The private part of this key is used to sign core secure world key certificate. The core secure world key certificate holds the hash of secure partition content certificate, and secure partition content certificate hold the hash of SPM binary (BL32).

Required certificates

The root of trust public key (ROTPK) will verify the CCA content certificate, and the platform root of trust public key (PROTPK) will verify the platform key certificate. All the firmware images are verified with the hash present in the respective parent certificate.

CCA content certificate

BL1 stage authenticate the signature of CCA content certificate using ROTPK. On success, the BL1 firmware loads BL2, calculate the BL2 image hash and compare the hash with the one present in the certificate. If hash match, then BL2 is a trusted binary and BL1 handover the execution to BL2.

The same way BL2 image loads BL31 and RMM images and verify the hash with CCA content certificate.

Platform key certificate

Platform key certificate is used to authenticate the non-trusted firmware content certificate.

Non-trusted firmware content certificate

Non-trusted firmware content certificate is used to authenticate the BL33 non-secure bootloader.

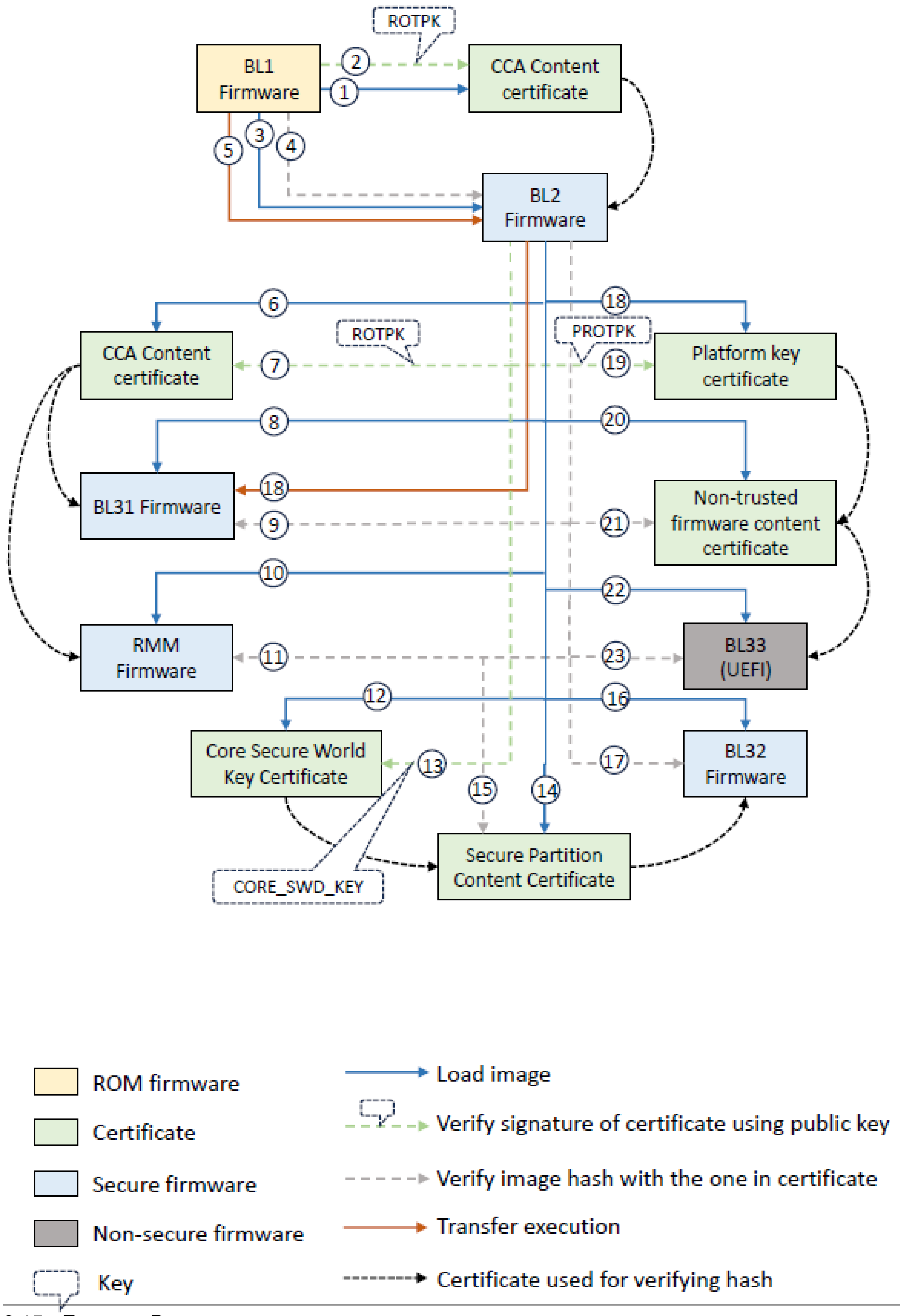
Core secure world key certificate

Core secure world key certificate is used to authenticate the secure partition content certificate.

Secure partition content certificate

Secure partition content certificate is used to authenticate the BL32 image.

Chain of trust authentication



Copyright (c) 2023, Arm Limited. All rights reserved.

2.15.12 Reliability, Availability, and Serviceability (RAS)

Reliability, Availability, and Serviceability (RAS)

Concept of RAS

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.

Overview

RAS spec divides the entire RAS architectural extension support into 2 into 2 categories

- ARMv8-A RAS Extension
- RAS system architecture

RAS architectural spec defines the hardware ras extensions the cpu and the system could implement to achieve desired level of RAS support. This document outlines concepts of RAS architecture important to understand the ras software architecture.

ARMv8-A RAS Extension define 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 define 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 for handling the given error e.g AP, SCP or MCP. This architectural definitions makes designing the software easier. Few component definitions that the RAS System architecture defines

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).

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 time stamp 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).

Software Error Handling

There are couple of approaches to achieve error handling in software. They are

- Firmware First Error Handling.
- Kernel First Error Handling.

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 spec 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.

Here are example platform implementations for Firmware First Error Handling.

- *Shared RAM ECC RAS support*
- *RdFremont CPU RAS support*

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 spec for RAS spec. 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.

Here is an example platform implementation for Kernel First Error Handling. *RdFremont CPU RAS support*

Error Injection Software

Error injeciton 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 design platform use the Error Injection (EINJ) ACPI table defined in the ACPI spec 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 platform 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 platform 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. The steps to exercise EINJ feature can be found in *Shared RAM ECC RAS support* and *RdFremont CPU RAS support*.

Copyright (c) 2024, Arm Limited. All rights reserved.

Shared RAM ECC RAS Test

Overview

The RD Fremont platform has 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 Fremont platform defines ECC RAS registers to log any ECC errors that occur during Shared RAM access from each master AP, SCP, MCP or RSS. There are 4 sets of ECC RAS registers defined for each master to log errors based on master's PAS. 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

For instance any error that occurs during SRAM access from AP when AP is executing in root PAS is logged into "AP Root RAM ECC RAS registers". This doc demonstrates the error logging for 1-bit CE that occurs during SRAM access from AP when executing in root PAS.

Note: This test is only supported on RD-Fremont-Cfg1 platform. The test is limited to error logging at EL3 and does not involve Host OS as explained in section "Firmware First Error Handling" of *RAS document*

1-bit CE 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.

Detailed Error injection software sequence is illustrated to inject 1-bit CE into Shared RAM from AP executing in root PAS.

- Add memory map for the Shared RAM ECC RAS registers memory space.
- Add memory map for the Shared memory space.
- **Program the SRAMECC_ERRMISC1 register to inject CE.**
    - **mmio_write_32((AP_RT_RAM_ECC_RAS_BASE + SRAM_ERR_MISC1_OFFSET),**
      SRAM_INJECT_ERROR_CE);
- **Read any Shared RAM region.**

- data = *(volatile uint32_t *) SHARED_RAM_ADDR

Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Procedure to perform 1-bit CE injection and handling on Shared RAM

Boot upto Busybox

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

Shared RAM error handling test

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
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.

```
INFO:    SGI: Base element RAM interrupt [85] handler
INFO:    ErrStatus = 0x86000000
INFO:    ErrAddr = 0x19100
```

Copyright (c) 2024, Arm Limited. All rights reserved.

Fremont CPU RAS Test

Overview

The RD Fremont platform has support for 2 error nodes.The presence of these nodes thus enables RAS extension on RD Fremont core.

- Node 0: Includes the L3 memory system in the DSU.
- Node 1: Includes the private L1 and L2 memory systems in the core.

The RAM's in the RD Fremont core support SED parity (Single Error Detect) and SECDED ECC (Single Error Correct Double Error Detect) capabilities.

RD Fremont also supports inserting errors in the error detection logic to verify error handling software.

Note: The RD Fremont platform is based on direct connect configuration and has no DSU. Hence RD Fremont reference design platform supports only one error node i.e Node0.

DE error injection on RD Fremont

Poseidon core implements Pseudo Fault Generation registers. With the help of these register software can inject either CE, DE or UE into the cache RAMs.

Detailed Error injection software sequence is illustrated to inject 1-bit DE.

- **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 Corrected Error injection write**
    - write_cpu_pfg_ctrl_register (0x80000020) // Generates FHI interrupt

Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Select the Build option

RD Fremont CPU supports both Firmware First and Kernel First Error handling. At givenpoint of time either of the support can be enabled. Firmware First Support is enabled by default. To enable Kernel First support disable build option TF_A_RAS_FW_FIRST=0. Navigate to your workspace and

- **For Firmware First**
    - vim build-scripts/configs/rdfremontcfg1/rdfremontcfg1
    - Set TF_A_RAS_FW_FIRST=1

- **For Kernel First**
  - vim build-scripts/configs/rdfremontcfg1/rdfremontcfg1
  - Set TF_A_RAS_FW_FIRST=0

Note: Clean and build once you switch error handling.

Procedure to perform DE injection and handling on N2 CPU

Boot upto Busybox

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

Fremont CPU error handling test

After the busybox boot is complete, use below commands to inject 1-bit DE on the RD Fremont. EINJ table debugfs enteries are used to inject the error. The "sel-firmware-first" field in oem-einj is utilized to toggle firmware first error injection, with the default being kernel first error injection. "sel-error-type" is used for choosing the type of error injection, and the current implementation suppports deferred errors.

- Firmware First Error Injection

```
mount -t debugfs none /sys/kernel/debug
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
```

- Kernel First Error Injection

```
mount -t debugfs none /sys/kernel/debug
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
```

Note: Error injection, whether firmware-first or kernel-first, are both initiated from the kernel.

Firmware First Error Handling

On successful error injection the firmware reception log's this error information on the console.

```
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
```

Kernel First Error Handling

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. Logs from kernel first error handling test.

```
[ 2365.760926] Injecting DE-
[ 2365.760928] ARM RAS: error from CPU7
[ 2365.760930] ERR0STATUS: 0x40800000
```

Copyright (c) 2022-2024, Arm Limited. All rights reserved.

SCP Error Injection Utility

RAS Error Injection Feature

Error injection feature is a micro-architecture feature defined by RAS to inject errors in the RAS supported system components. Components, supporting RAS, have registers for Error injection. Software can use these registers to inject an error and verify the error handling software.

Overview of Error Injection 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, subcomponents represent error injection in different worlds: Root, Secure, Realm, and Non-Secure. “Type” defines the various types of errors supported by each component.

Procedure to inject errors into various components

Boot upto SCP CLI Debugger Shell

- 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.
- To exit the CLI Debugger, press Ctrl + D.

  > 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:**
     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

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 various error injection scenarios

Component, Subcomp, Type of Error	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 [RSM_INT] ErrAddr = 0x10
RSM SRAM Non-secure World UE einj-util -comp 3 -subcomp 1 -type 1	Injecting UE into RSM SRAM [RSM_INT] ErrStatus = 0xa4000000 [RSM_INT] fwk_int number = 29 [RSM_INT] ErrAddr = 0x10
TCM ITCM CE einj-util -comp 1 -subcomp 0 -type 0	ITCM Injecting CE [TCM_INT] ErrStatus = 0x5 [TCM_INT] fwk_int number = 21 [TCM_INT] ErrCode = 0x9 [TCM_INT] ErrAddr = 0x6b38 Data : 0x20040000
TCM ITCM UE einj-util -comp 1 -subcomp 0 -type 1	[TCM_INT] ErrStatus = 0x7 [TCM_INT] fwk_int number = 21 [TCM_INT] ErrCode = 0x9 [TCM_INT] ErrAddr = 0x6a46 ITCM Injecting UE Data : 0x20040000
TCM DTCM CE einj-util -comp 1 -subcomp 1 -type 0	DTCM Injecting CE [TCM_INT] ErrStatus = 0x7 [TCM_INT] fwk_int number = 21 [TCM_INT] ErrCode = 0xb [TCM_INT] ErrAddr = 0x6b3c Data : 0xFFFFFFFE
TCM DTCM UE einj-util -comp 1 -subcomp 1 -type 1	[TCM_INT] ErrStatus = 0x7 [TCM_INT] fwk_int number = 21 [TCM_INT] ErrCode = 0xb [TCM_INT] ErrAddr = 0x6a46 DTCM Injecting UE Data : 0xFFFFFFFE
CPU Core CE einj-util -comp 2 -subcomp 0 -type 0	Injecting CPU CE ErrStatus 0xC6000000 ErrAddress 0x0
CPU Core UE einj-util -comp 2 -subcomp 0 -type 1	Injecting CPU UE ErrStatus 0x60000000 ErrAddress 0x0
CPU Core DE einj-util -comp 2 -subcomp 0 -type 2	Injecting CPU DE ErrStatus 0x40800000 ErrAddress 0x0

Copyright (c) 2024 Arm Limited. All rights reserved.

2.16 Fremont-Cfg2 Documents

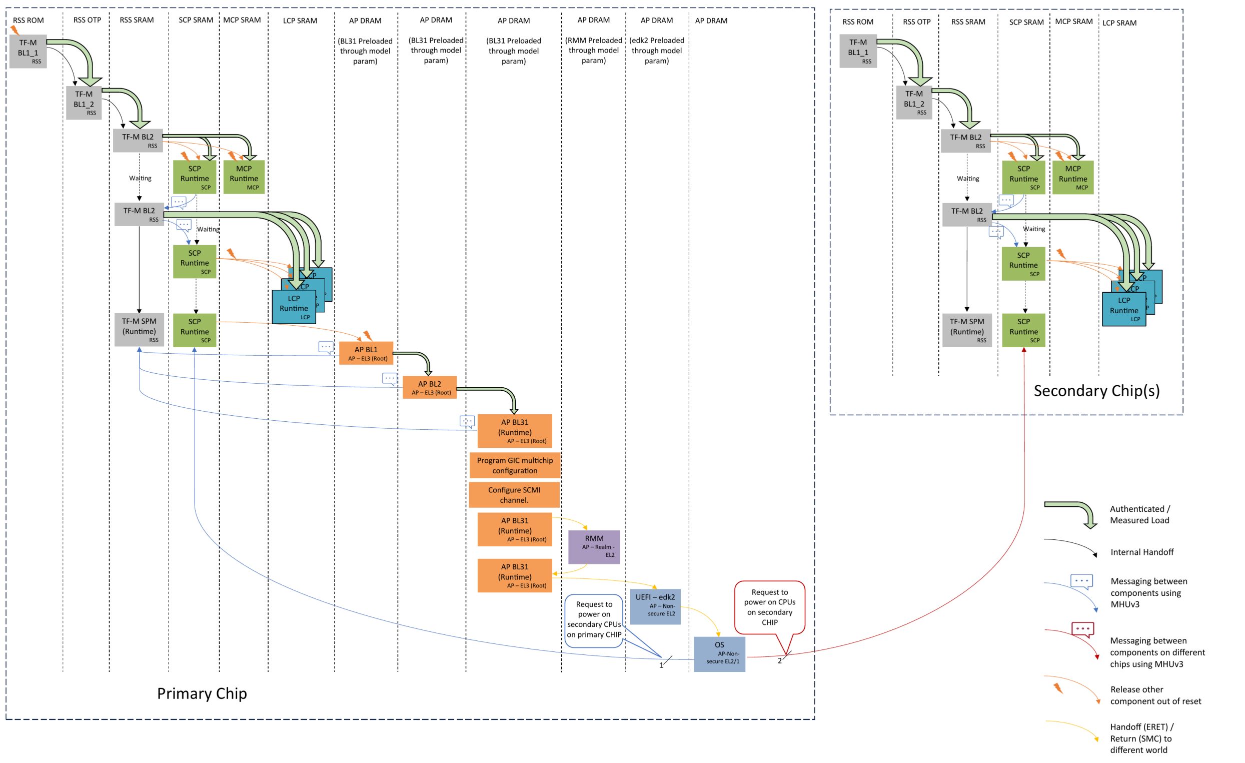
2.16.1 Boot Flow for RD-Fremont-Cfg2 platform

Introduction

This page describes the overview of the software boot process on RD-Fremont-Cfg2 platform variant.

Overview

A simplified boot flow diagram is shown below.


RSS ROM
RSS OTP
RSS SRAM
SCP SRAM
MCP SRAM
LCP SRAM
AP DRAM
(BL31 Preloaded through model param)
AP DRAM
(BL31 Preloaded through model param)
AP DRAM
(BL31 Preloaded through model param)
AP DRAM
(RMM Preloaded through model param)
AP DRAM
(edk2 Preloaded through model param)
AP DRAM
TF-M BL1_1 RSS
TF-M BL1_2 RSS
TF-M BL2 RSS
SCP Runtime SCP
MCP Runtime MCP
Waiting
TF-M BL2 RSS
Waiting
SCP Runtime SCP
LCP Runtime LCP
TF-M SPM (Runtime) RSS
SCP Runtime SCP
AP BL1 AP – EL3 (Root)
AP BL2 AP – EL3 (Root)
AP BL31 (Runtime) AP – EL3 (Root)
Program GIC multichip configuration
Configure SCMI channel.
AP BL31 (Runtime) AP – EL3 (Root)
RMM AP – Realm - EL2
AP BL31 (Runtime) AP – EL3 (Root)
UEFI – edk2 AP – Non-secure EL2
Request to power on secondary CPUs on primary CHIP
Request to power on CPUs on secondary CHIP
OS AP-Non-secure EL2/1
1
2
Primary Chip
RSS ROM
RSS OTP
RSS SRAM
SCP SRAM
MCP SRAM
LCP SRAM
TF-M BL1_1 RSS
TF-M BL1_2 RSS
TF-M BL2 RSS
SCP Runtime SCP
MCP Runtime MCP
Waiting
TF-M BL2 RSS
Waiting
SCP Runtime SCP
LCP Runtime LCP
TF-M SPM (Runtime) RSS
SCP Runtime SCP
Secondary Chip(s)
Authenticated / Measured Load
Internal Handoff
Messaging between components using MHUv3
Messaging between components on different chips using MHUv3
Release other component out of reset
Handoff (ERET) / Return (SMC) to different world

RSS

On platform reset, RSS on each chip are released out of reset and SCP and MCP are held in reset state. RSS on each chip begin executing the TF-M's BL1_1 from RSS ROM and provision the BL1_2 image into the One Time Programmable flash (OTP) of the respective chip and transfers the execution to the BL1_2 stage. More details on the provisioning can be found in the TF-M's RSS provisioning page. BL1_2 authenticates and loads the TF-M's BL2 (MCUBoot) stage which is responsible for authenticated loading of the next stage images as well as images of the other components in the respective chip. More details on BL2 can be found in *Image Loading Using MCUBoot* page. BL2 stage is also responsible for *setting up the SCP's ATU*, *NI-Tower* instances and releasing SCP and MCP out of reset.

SCP

SCP is responsible for managing the per chip system power, setting up of the interconnect and configuring the interconnect for cross chip communication. More details on the interconnect setup can be found in *CMN-Cyprus Driver Module* and in *CMN-Cyprus Multichip Configuration* page. SCP is also responsible for releasing the LCPs out of reset after RSS loads the LCP firmware into ITCMs. RSS has to wait for the SCP to turn on the SYSTOP power domain before loading the LCP images. To maintain this synchronization, SCP and RSS communicates messages through MHUv3. More details on this can be found in *SCP - RSS Communication* page. Only the SCP on the primary chip releases the primary core out of reset. At this point, SCP on the secondary chips will wait for the SCMI messages.

LCP

LCP is responsible for managing per Application Processor's power. The boot sequence of LCP is summarized in *Local Control Processor* page.

Copyright (c) 2023, Arm Limited. All rights reserved.

2.16.2 CMN-Cyprus Multichip Configuration

Introduction

This document gives an overview of the CMN-Cyprus multichip configuration in the RD-Fremont-Cfg2 platform.

Overview

The following diagram shows the cross-chip CCG connections in the the RD-Fremont-Cfg2 platform:

```
+----------------------------------+             +------------------------------
↪--+
|                                  |             |                             ␣
↪   |
|                              CCG6+-------------+CCG6                         ␣
↪   |
|                                  |             |                           ␣
↪CCG4|
|CCG4                          CCG7+-------------+CCG7                         ␣
↪   |
|             CHIP 0               |             |             CHIP 2          ␣
↪   |
|                              CCG8+-----+  +-----+CCG8                       ␣
↪CCG5|
|CCG5                              |    |  |     |                             ␣
↪   |
|                              CCG9+--+ |  |  +--+CCG9                         ␣
↪   |
|                                  |  | |  |  |  |                             ␣
↪   |
|   CCG0    CCG1    CCG2    CCG3   |  | |  |  |  |  CCG0  CCG1  CCG2  CCG3␣
↪   |
+----+-------+-------+-------+-----+  | |  |  |  +-----+------+------+------+--
↪--+
```

(continues on next page)

(continued from previous page)

```
     |       |       |       |       |  |  |  |        |      |      |      |
     |       |       |       |       |  |  |  |        |      |      |      |
+----+-------+-------+-------+-----+ |  |  |  |  +-----+------+------+------+--
↪--+
|   CCG3    CCG2    CCG1    CCG0   | |  |  |  |  |    CCG3  CCG2   CCG1   ␣
↪CCG0   |
|                                  | |  |  |  |  |                            ␣
↪   |
|                              CCG4+-|--|--+  |  |                            ␣
↪   |
|                                  | |  |     |  |                            ␣
↪   |
|                              CCG5+-|--|-----+  |                            ␣
↪   |
|                                  | |  |        |                            ␣
↪   |
|CCG8                              | |  +--------+CCG4                      ␣
↪CCG8|
|             CHIP 1               | |           |             CHIP 3         ␣
↪   |
|                                  | +-----------+CCG5                        ␣
↪   |
|CCG9                              |             |                          ␣
↪CCG9|
|                              CCG6+-------------+CCG6                        ␣
↪   |
|                                  |             |                            ␣
↪   |
|                              CCG7+-------------+CCG7                        ␣
↪   |
|                                  |             |                            ␣
↪   |
+----------------------------------+             +-----------------------------
↪--+
```

CMN-Cyprus Driver config data for multichip

Please refer the following file in the workspace to get the complete info on the configuration data passed to the CMN-Cyprus driver module in SCP firmware:

`<workspace>/scp/product/neoverse-rd/rdfremont/scp_ramfw/config_cmn_cyprus.c`

The CML configuration structures can be found in the following file: `<workspace>/scp/module/cmn_cyprus/include/mod_cmn_cyprus.h`

The `mod_cmn_cyprus_cml_config` structure is used to describe the CCG blocks. Each chip in the RD-Fremont-Cfg2 platform is connected to the other chips via CCG blocks. This info is described using the following tables in the config data:

- `cml_config_table_chip_0`: This table describes the CCG blocks that connect the Chip 0 to Chip 1, Chip 2 and Chip 3 respectively.
- `cml_config_table_chip_1`: This table describes the CCG blocks that connect the Chip 1 to Chip 0, Chip 2 and Chip 3 respectively.

- `cml_config_table_chip_2`: This table describes the CCG blocks that connect the Chip 2 to Chip 0, Chip 1 and Chip 3 respectively.
- `cml_config_table_chip_3`: This table describes the CCG blocks that connect the Chip 3 to Chip 0, Chip 1 and Chip 2 respectively.

The following section explains how the Chip 0 CCG blocks are described in the `cml_config_table_chip_0` table. **Please note that LCN is enabled along with CPAG and two CCG nodes per CPAG group is considered in this config data.**

1. Configure the logical IDs for the CCGs. The logical ID ranges from 0 to (total CCG block per chip - 1).
    - Refer `rdfremontcfg2_cmn_cyprus_ccg_port` in config_cmn_cyprus.c file for the list of CCG logical IDs.
    - For the CCG blocks that connect Chip 0 to Chip 1, `CCG_PORT_0` and `CCG_PORT_1` is used.
2. Configure HAIDs for the CCG blocks. This is a unique HAID across all the chips.
    - The Chip ID is used to calculate a unique HAIDs for the CCG blocks. Hence, `(CCG_PER_CHIP * PLATFORM_CHIP_0) + CCG_PORT_0` and `(CCG_PER_CHIP * PLATFORM_CHIP_0) + CCG_PORT_1` is used.
3. Configure the table of remote memory regions. As LCN is enabled, the remote memory regions must be specified as System Cache Group(SCG) regions. The remote DRAM regions which are non-contiguous are specified as secondary regions for the same SCG. Apart from these, the start and the end node positions for the remote SCG is also configured.
    - Since the CCG0, CCG1 blocks in Chip 0 is connected to Chip 1, the address range of the Chip 1 is specified in this table: i.e., [0x1000000000 - 0x1ffffffff] and [0x100000000000 - 0x1ffffffffff]
    - Specify the unique HAIDs of the CCG3 and CCG2 blocks in Chip 1 as the target HAIDs.
4. Configure the Remote Chip ID. This is used to setup the AgentID (RAID and HAID) to LinkID LUT.
5. Set `enable_smp_mode` to true as the CCG block must be programmed for multichip SMP communications.
6. Set `enable_direct_connect_mode` to true in order to enable the upper link layer to upper link layer connection between the CCLAs of two chips. This enables the connection of CXS interface from the CCLA on one CMN-Cyprus to the CXS interface of the other.
7. Set `enable_cpa_mode` to true in order to enable CCIX Port Aggregation Mode.
8. Set `cpag_id` to 0, as the CPA group IDs are assigned sequentially and `ccg_count` to 2 as we consider two CCG ports per CPA group.

Steps 1 - 8 are repeated for configuring the CPAG that connect Chip 0 to Chip 2 and Chip 3 respectively. Similarly, the CPA groups in Chip 1, 2 and 3 are specified in the respective CCG config tables. Also, the `enable_lcn` flag is set in each chip's CMN config data to enable LCN programming.

The CMN-Cyprus driver in each SCP, discovers and configures the local CMN mesh and then programs the CCGs to enable multichip SMP communication. For more info on the CMN-Cyprus driver programming, please refer *CMN-Cyprus Driver*.

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

2.16.3 RD-Fremont multichip memory map

Introduction

RD-Fremont-Cfg2 is a quad chip platform. One of the chip is identified as primary chip and the other three as secondary chip. Each chip is allocated a different address region in the system memory map. The CMN interconnect on each chip is programmed with its own RN-SAM at boot time.

- The PCIe ECAM/MMIO regions are split across multiple chips by configuring the CMN RN-SAM.
- The DRAM memory regions are split across multiple chips by configuring the CMN RN-SAM.

Multichip memory map

For RD-Fremnt-Cfg2 platform, the per chip address size for peripherals is 64GB, and the base address for peripherals residing in each chip are:

Start Addr	End Addr	Size	Chiplet
0x00_0000_0000	0x0F_FFFF_FFFF	64 GB	Chip0 Onchip Peripheral space base address
0x10_0000_0000	0x1F_FFFF_FFFF	64 GB	Chip1 Onchip Peripheral space base address
0x20_0000_0000	0x2F_FFFF_FFFF	64 GB	Chip2 Onchip Peripheral space base address
0x30_0000_0000	0x3F_FFFF_FFFF	64 GB	Chip3 Onchip Peripheral space base address
0x40_0000_0000	0xFFFF_FFFF_FFFF	255.75 TB	Can be mapped to any chiplet

Multichip interrupt map

On the shared peripheral interrupt (SPI), each chip are assigned with 480 SPIs. The SPI mappings are:

Chiplet	SPI ID mapping
Chip0	32 - 511
Chip1	512 - 991
Chip2	4096 - 4575
Chip3	4576 - 5055

Multichip DRAM map

Each chip of RD-Fremont-Cfg2 FVP is equiped with two DRAM blocks, a 2GB and a 6GB. The 6GB DRAM of the secondary chips are kept outside of the 64GB address per chip. The DRAM mappings are:

Chiplet	Size	Start Addr	End Addr
Chip0	2GB	0x0000_8000_0000	0x0000_FFFF_FFFF
Chip0	6GB	0x0080_8000_0000	0x0081_FFFF_FFFF
Chip1	2GB	0x0010_8000_0000	0x0010_FFFF_FFFF
Chip1	6GB	0x2080_8000_0000	0x2081_FFFF_FFFF
Chip2	2GB	0x0020_8000_0000	0x0020_FFFF_FFFF
Chip2	6GB	0x3080_8000_0000	0x3081_FFFF_FFFF
Chip3	2GB	0x0030_8000_0000	0x0030_FFFF_FFFF
Chip3	6GB	0x4080_8000_0000	0x4081_FFFF_FFFF

Copyright (c) 2023, Arm Limited. All rights reserved.

CHAPTER

THREE

PLATFORM BOOT

3.1 Busybox boot

3.1.1 Overview of busybox boot

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 compontents 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.

3.1.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

3.1.3 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 parition 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 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 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
```

3.1.4 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"
```

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

3.2 Buildroot boot

3.2.1 What is Buildroot?

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.

3.2.2 Overview of Buildroot Boot

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.

3.2.3 Sync the required platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

3.2.4 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 parition 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 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 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
```

3.2.5 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
  ```

3.2.6 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 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 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"
```

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

3.3 Install and boot a Linux distribution

3.3.1 Linux distribution boot

Neoverse Reference Design (RD) platform software stack supports the installation and boot of various linux distributions such as Debian, Ubuntu or Fedora. The distribution is installed on a SATA disk and the installed image can be used for multiple boots.

3.3.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

3.3.3 Build the platform software

This section describes the procedure to build the platform firmware required to install and boot 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)

Examples of the build command are

- Command to clean, build and package the RD-N2 software stack required for the distro installation/boot test on the 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-V1 quad-chip platform:

  ```
  ./build-scripts/build-test-uefi.sh -p rdv1mc 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 rdv1mc package
  ```

3.3.4 Installing a linux distribution

After the build of the platform software stack for the distro install/boot 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.

Latest version of distro installation images can be downloaded from the following locations (select an image built for aarch64 architecture):

- Fedora
- Ubuntu

- Debian

The commands used to begin the distro installation 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 installation:

```
./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.

An example of a command to install the debian distribution is listed below.

```
./distro.sh -p rdv1 -i /absolute/path/to/debian-10.6.0-arm64-xfce-CD-1.iso -s 16
```

- This command creates a 16GB SATA disk image, boots the RD-V1 single-chip software stack and starts the installation of debian distribution.
- From here on, follow the instructions of the choosen distribution installer. For more information about the installation procedure, refer online installation manuals of the choosen distribution.
- After the installation is completed, the 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 Distribution installation:
  - 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 displays 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.

3.3.5 Booting a linux distribution

To boot the installed distro, use the following commands:

- 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
```

- Start the distro boot:

```
./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 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.

Example commands to boot a linux distribution are listed below.

- Command to look for the available .satadisk image in the `model-scripts/rdinfra` folder and boots with that image. If multiple `.satadisk` images are found, it will list them all but won't boot:

```
./distro.sh -p rdv1
```

- Command to begin the distro boot from the `fedora.satadisk` image:

```
./distro.sh -p rdv1 -d /absolute/path/to/fedora.satadisk
```

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

3.4 UEFI Secure Boot

3.4.1 What is UEFI Secure Boot

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.

3.4.2 Secure boot support for RD platforms

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.

3.4.3 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

3.4.4 Generate key pairs

The next step is to create key pairs required for secure boot. The one-time generation of the following key pairs are mandatory - PK, KEK, DB and DBX. The following commands can be used to generate these key pairs.

Note: Execute the following commands from the workspace into which the platform software has been downloaded.

- 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.

3.4.5 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
```

3.4.6 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 atleast 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.

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

3.5 WinPE boot

3.5.1 WinPE boot

Neoverse Reference Design (RD) platform software stack supports the boot of Windows Preinstallation 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.

3.5.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

3.5.3 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
```

3.5.4 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.

3.5.5 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
```

Copyright (c) 2020-2023, Arm Limited. All rights reserved.

CHAPTER

FOUR

TEST SUITES

4.1 ACS compliance test

4.1.1 What is Arm ACS

Architecture Compliance Suite (ACS) is used to ensure architectural compliance across different implementations of the architecture. Arm Enterprise ACS includes a set of examples of the invariant behaviours that are provided by a set of specifications for enterprise systems (e.g. SBSA, SBBR, etc.), so that implementers can verify if these behaviours have been interpreted correctly.

4.1.2 Overview of ACS test

Reference design (RD) platform is targeted for enterprise, network and other infrastructure markets. This requires the platforms to be compliant to the various architectural specifications such as SBSA and SBBR. ACS compliance test helps to ensure that the RD platform and the software stack is compliant to these specifications. The ACS compilance test on RD platforms use a downloadable pre-built ACS disk image. On completion of the ACS test on RD platforms, the test results are stored on the ACS disk image and can be retrieved from it. The results can then be looked into to determine the conformance to the SBSA and SBBR standards.

4.1.3 Download the required platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

4.1.4 Build the platform software

This section describes the procedure to build the RD software for ACS test. The components of the RD platform software stack that are built is limited to those that provide the EFI implementation and the EFI shell (i.e, SCP, TF-A and EDK2).

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

- If the target platform is SGI-575:

```
./build-scripts/sgi/build-test-acs.sh -p sgi575 <command>
```

- If the target platform is a Reference Design (RD) platform:

```
./build-scripts/rdinfra/build-test-acs.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 ACS test on RD-N2 platform:

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

4.1.5 Validate ACS conformance

For running the ACS tests, the ACS test suite disk image is required. The ACS test suite disk image can either by built from source by following the documentation at SystemReady SR ACS or the latest available prebuilt image (sr_acs_live_image.img.xz) can be downloaded from here and extract the sr_acs_live_image.img from sr_acs_live_image.img.xz as described in the README file. It is advised that the latest prebuilt image be used if there are no specfic reasons to build this disk image from ACS test suit source code.

Note: The fresh copy of the ACS test suite disk image should be used before starting the ACS tests. This is to ensure that the tests are not skipped due to presence of log files in the disk image from the previous execution of the test.

To boot and to start ACS test, the commands to be used are

- Set `MODEL` path before launching the model:

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

- If the target platform is SGI-575:

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

- If the target platform is a Reference Design (RD) platform

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

- Launch the ACS test

```
./acs.sh -p <platform name> -v <path to sr_acs_live_image.img> -n [true|false] -a
↪<additional_params>
```

Supported command line options are listed below

- -p <platform name>
  - Lookup for a platform name in *Platform Names*.
- -v <absolute path to the sr_acs_live prebuilt image>
  - The absolute path to the sr_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.

Example commands to perform the ACS tests are as listed below.

- Command to start the execution of the RD-N2 model and perform the ACS tests. The ACS test suite disk image named 'sr_acs_live_image.img' is picked up from the location /tmp/sr_acs_live_image.img.

```
./acs.sh -p rdn2 -v /tmp/sr_acs_live_image.img
```

Note: Follow the instructions below for the steps to be performed to complete the tests.

- Command to start the execution of the RD-N2 model with networking enabled and perform the ACS tests. The ACS test suite disk image named 'sr_acs_live_image.img' is picked up from the location /tmp/sr_acs_live_image.img. Additional parameters to the model are supplied using the -a command line parameter and networking support is enabled by using the -n parameter.

```
./acs.sh -p rdn2 -v /tmp/sr_acs_live_image.img -n true -a "-C board.flash0.
↪diagnostics=1"
```

Note: Follow the instructions below for the steps to be performed to complete the tests.

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 ACS test cases, choose the option 'bbr/bsa' from Grub menu, which will launch the EFI shell. Press 'Enter' key or wait till the timeout for the EFI startup script to run. The startup script will launch SBBR SCT test by default at first, skip the SCT test by pressing any key on the prompt 'Press any key to stop the EFI SCT running'. This will start the SBSA UEFI test to start (if SCT is not skipped, the SBSA will run after completion of SCT).

On UEFI SBSA test completion, the startup script will reboot the platform, follow the same steps mentioned above till skipping SCT. This time SBSA UEFI test is not executed as the results are already captured in the sr_acs_live_image.img disk image and the validation OS starts boot. The Linux part of the test will be executed on validation OS boot complete.

On completion of the ACS SBSA and SBBR tests, the execution stops at the sr_acs_live_image filesystem command line prompt. The test can be stopped by terminating the FVP.

In case it is not required to run the complete ACS compliance, instead it is required to validate only the SBSA, the sr_acs_live_image.img has the provision. For this the SBSA test should be run from the EFI shell manually by executing the command listed below. This command skips the exerciser tests as well.

```
Shell> EFI\BOOT\bsa\sbsa\Sbsa.efi -skip 800
```

Running the test manually will not store the test result into sr_acs_live_image.img disk image, instead the test results will be available on console as the test proceeds to completion.

4.1.6 Select a SBSA compliance level (optional)

SBSA specification classifies hardware into different levels, level-3 through level-7. The ACS disk image is typically configured for a default level. For ACS disk image v3.0, the default SBSA level is 4. For running the ACS tests for any higher or lower level, press ESC from UEFI shell and run the SBSA efi binary manually to select the appropriate compliance level to be tested. An example of the command to be used to select the compliance level is listed below.

```
Shell> EFI\BOOT\bsa\sbsa\Sbsa.efi -l <y>

Here, y can be 3, 4, 5 or 6 for the SBSA compliance level.
```

4.1.7 Retrieve the test results

On completion of SBSA/SBBR tests, the test results can be retrieved by mounting the second partition of ACS test suite disk image that was used for the test.

```
mkdir /tmp/acs-disk
sudo mount -o loop,offset=537919488 sr_acs_live_image.img /tmp/acs-disk/
```

The test results can be found in the directories below:

- UEFI SBSA test report: /tmp/acs-disk/acs_results/uefi/
- Linux SBSA test report : /tmp/acs-disk/acs_results/linux/
- FWTS result : /tmp/acs-disk/acs_results/fwts/

Unmount the disk after analysing the logs using the following commands.

```
losetup
sudo umount /dev/loop_x     # _x can be 0, 1, 2... based on losetup output
```

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

4.2 TF-A-Tests boot

4.2.1 Overview of tf-a-tests boot

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.

4.2.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

4.2.3 Build the platform software

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
```

4.2.4 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
> 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
```

(continues on next page)

(continued from previous page)

```
> 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
Tests Passed  : 60
Tests Failed  : 0
Tests Crashed : 0
Total tests   : 182
=================================
NOTICE:  Exiting tests.
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

4.3 SCT Standalone test

4.3.1 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.

4.3.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

4.3.3 Build the platform software

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 parition 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
```

4.3.4 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:

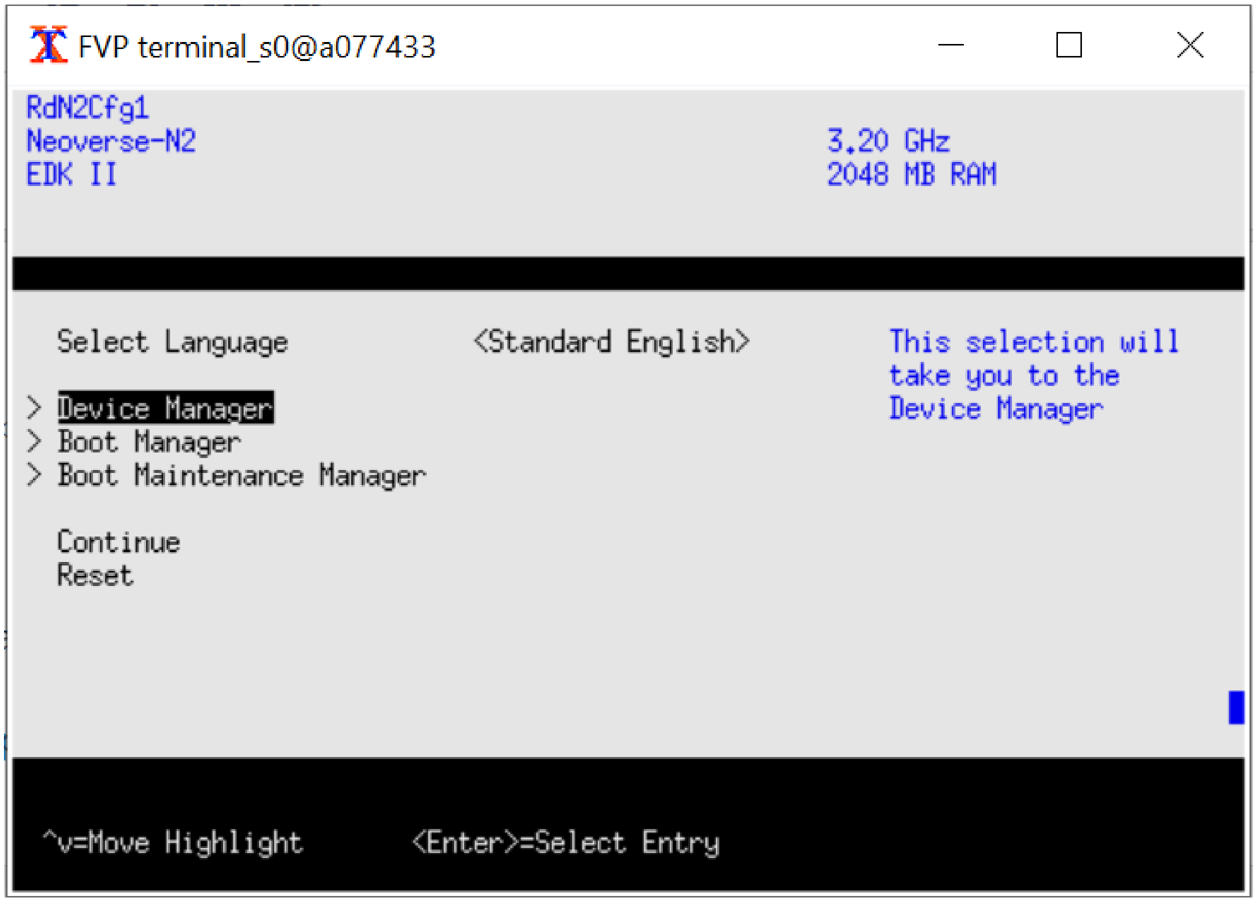
```
./sct.sh -p rdn2
```

- 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
```

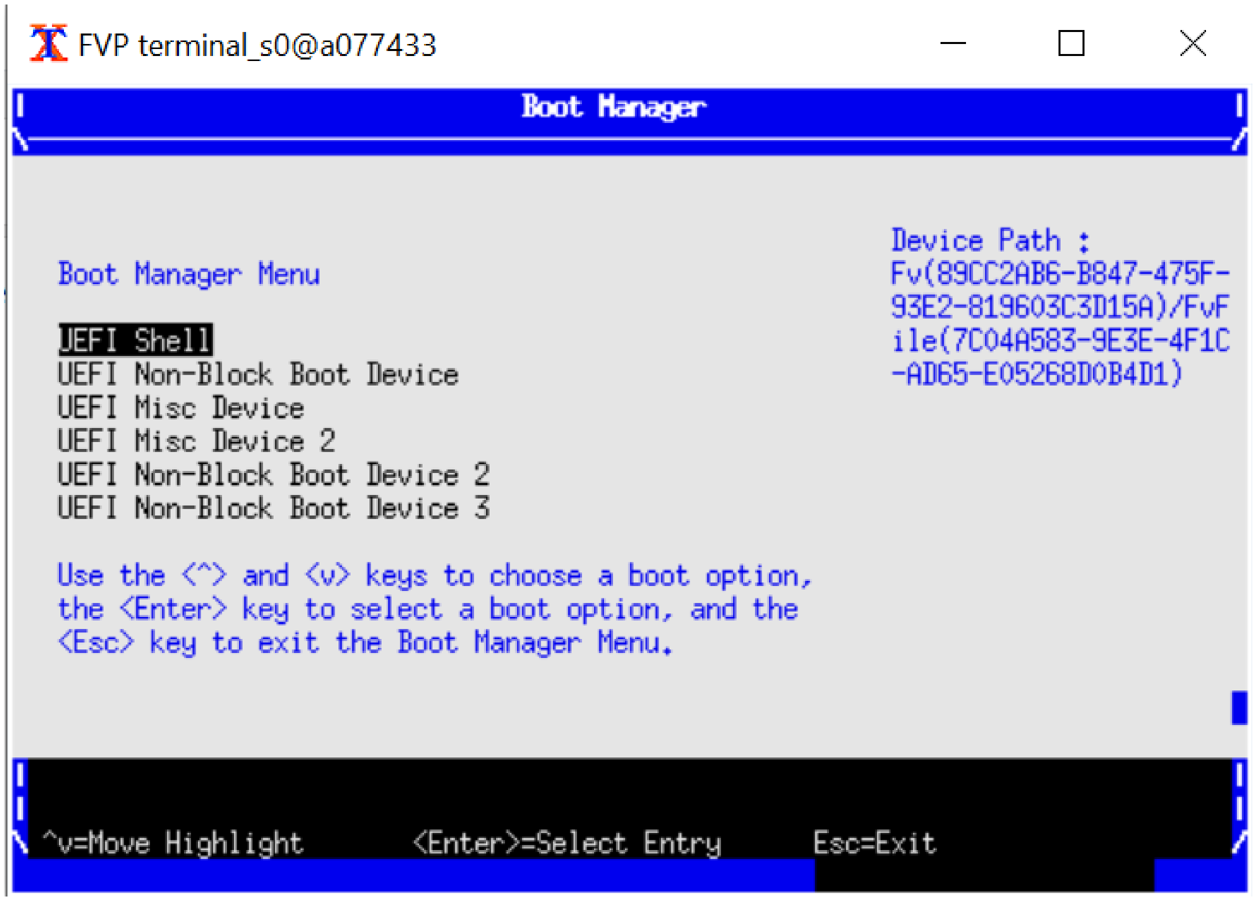
- There are additional steps to be performed on the first boot to run SCT test. These steps are listed below.

1- Click ESC and click on "Boot Manager"



```
FVP terminal_s0@a077433

RdN2Cfg1
Neoverse-N2                                          3.20 GHz
EDK II                                               2048 MB RAM

  Select Language             <Standard English>     This selection will
                                                     take you to the
> Device Manager                                     Device Manager
> Boot Manager
> Boot Maintenance Manager

  Continue
  Reset

 ^v=Move Highlight       <Enter>=Select Entry
```

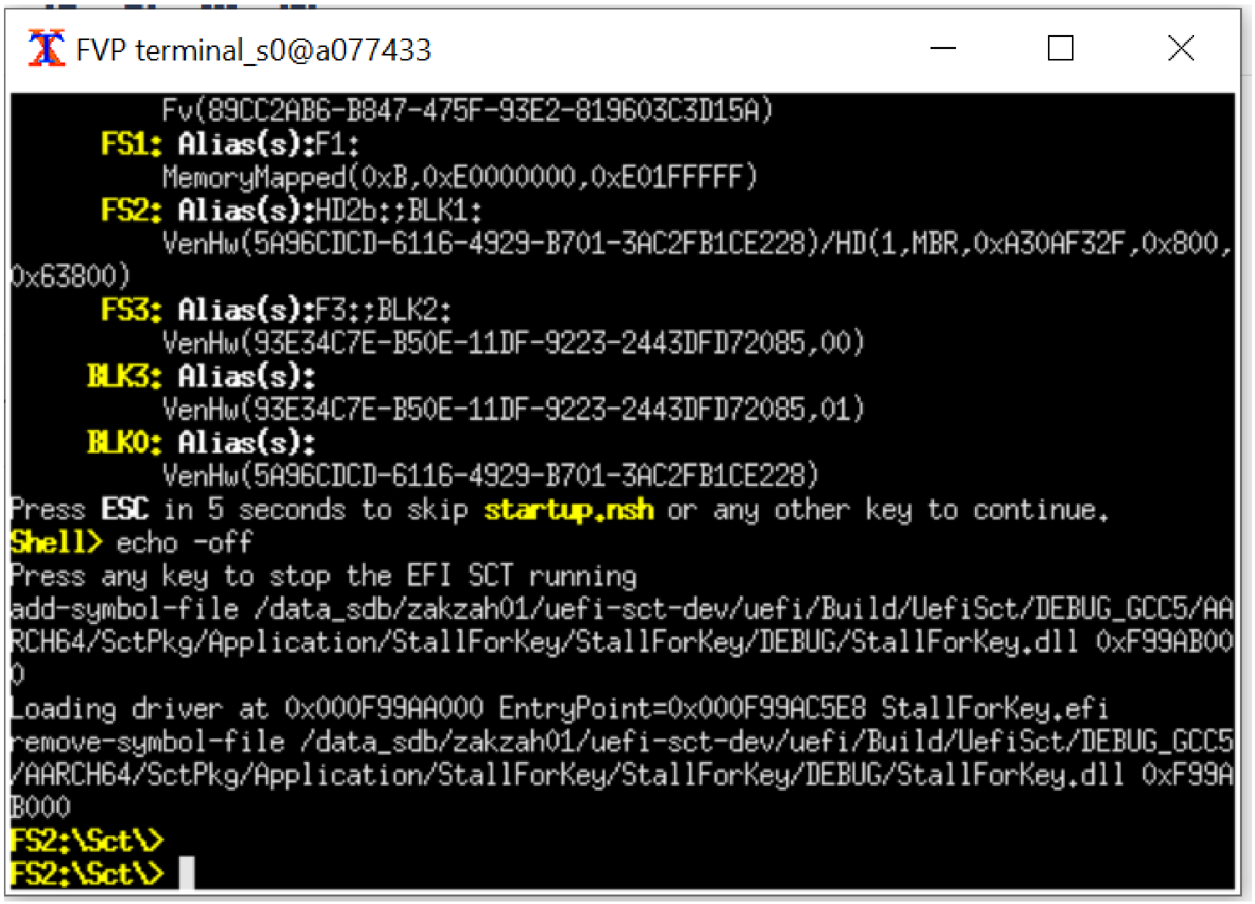
2- Select UEFI Shell and click Enter



```
FVP terminal_s0@a077433

                           Boot Manager

                                                   Device Path :
  Boot Manager Menu                                Fv(89CC2AB6-B847-475F-
                                                   93E2-819603C3D15A)/FvF
  UEFI Shell                                       ile(7C04A583-9E3E-4F1C
  UEFI Non-Block Boot Device                       -AD65-E05268D0B4D1)
  UEFI Misc Device
  UEFI Misc Device 2
  UEFI Non-Block Boot Device 2
  UEFI Non-Block Boot Device 3

  Use the <^> and <v> keys to choose a boot option,
  the <Enter> key to select a boot option, and the
  <Esc> key to exit the Boot Manager Menu.

 ^v=Move Highlight       <Enter>=Select Entry      Esc=Exit
```

3- In the UEFI Shell tap the command to run the SCT test navigator

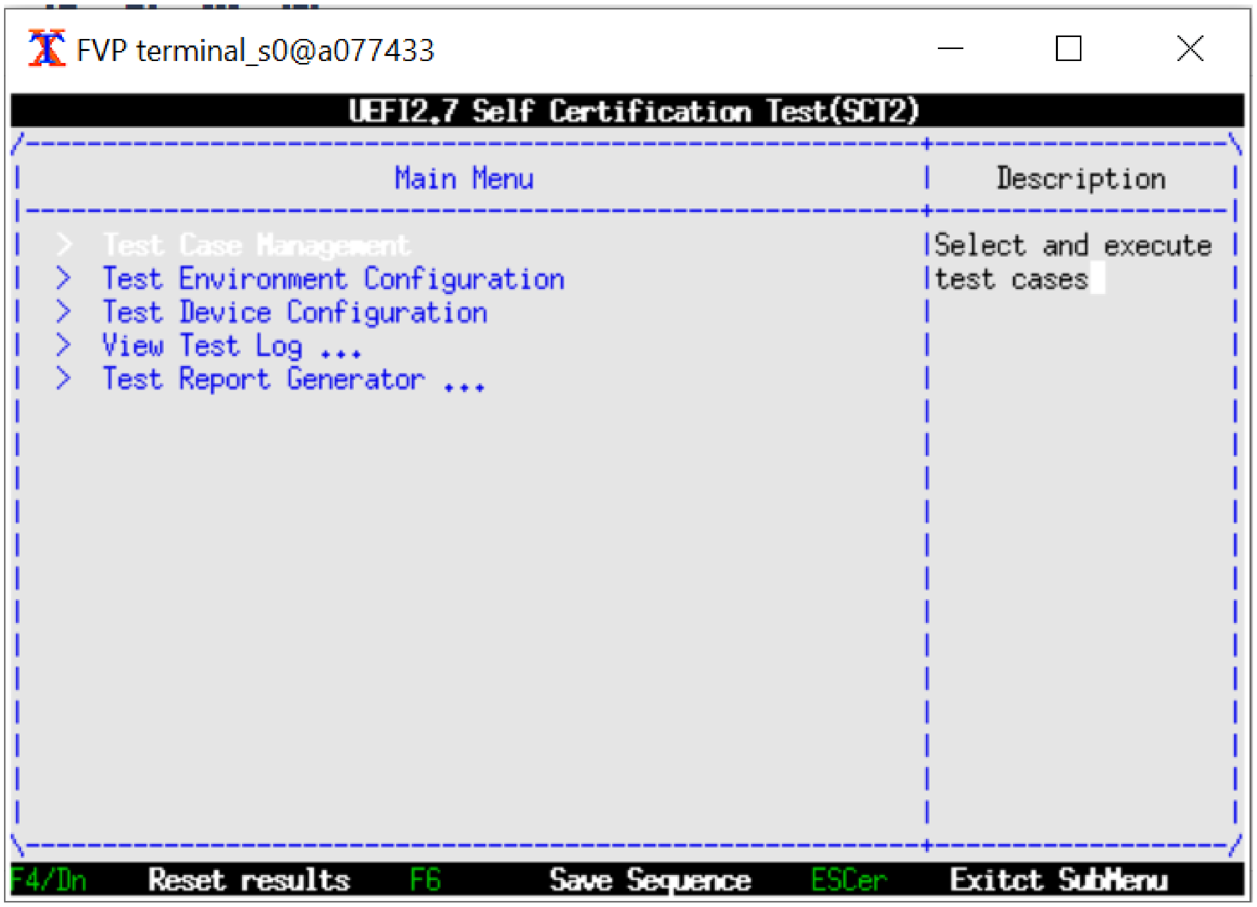
```
SCT.efi -u
```



FVP terminal_s0@a077433

```
          Fv(89CC2AB6-B847-475F-93E2-819603C3D15A)
      FS1: Alias(s):F1:
          MemoryMapped(0xB,0xE0000000,0xE01FFFFF)
      FS2: Alias(s):HD2b::BLK1:
          VenHw(5A96CDCD-6116-4929-B701-3AC2FB1CE228)/HD(1,MBR,0xA30AF32F,0x800,
0x63800)
      FS3: Alias(s):F3::BLK2:
          VenHw(93E34C7E-B50E-11DF-9223-2443DFD72085,00)
     BLK3: Alias(s):
          VenHw(93E34C7E-B50E-11DF-9223-2443DFD72085,01)
     BLK0: Alias(s):
          VenHw(5A96CDCD-6116-4929-B701-3AC2FB1CE228)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> echo -off
Press any key to stop the EFI SCT running
add-symbol-file /data_sdb/zakzah01/uefi-sct-dev/uefi/Build/UefiSct/DEBUG_GCC5/AA
RCH64/SctPkg/Application/StallForKey/StallForKey/DEBUG/StallForKey.dll 0xF99AB00
0
Loading driver at 0x000F99AA000 EntryPoint=0x000F99AC5E8 StallForKey.efi
remove-symbol-file /data_sdb/zakzah01/uefi-sct-dev/uefi/Build/UefiSct/DEBUG_GCC5
/AARCH64/SctPkg/Application/StallForKey/StallForKey/DEBUG/StallForKey.dll 0xF99A
B000
FS2:\Sct\>
FS2:\Sct\>
```

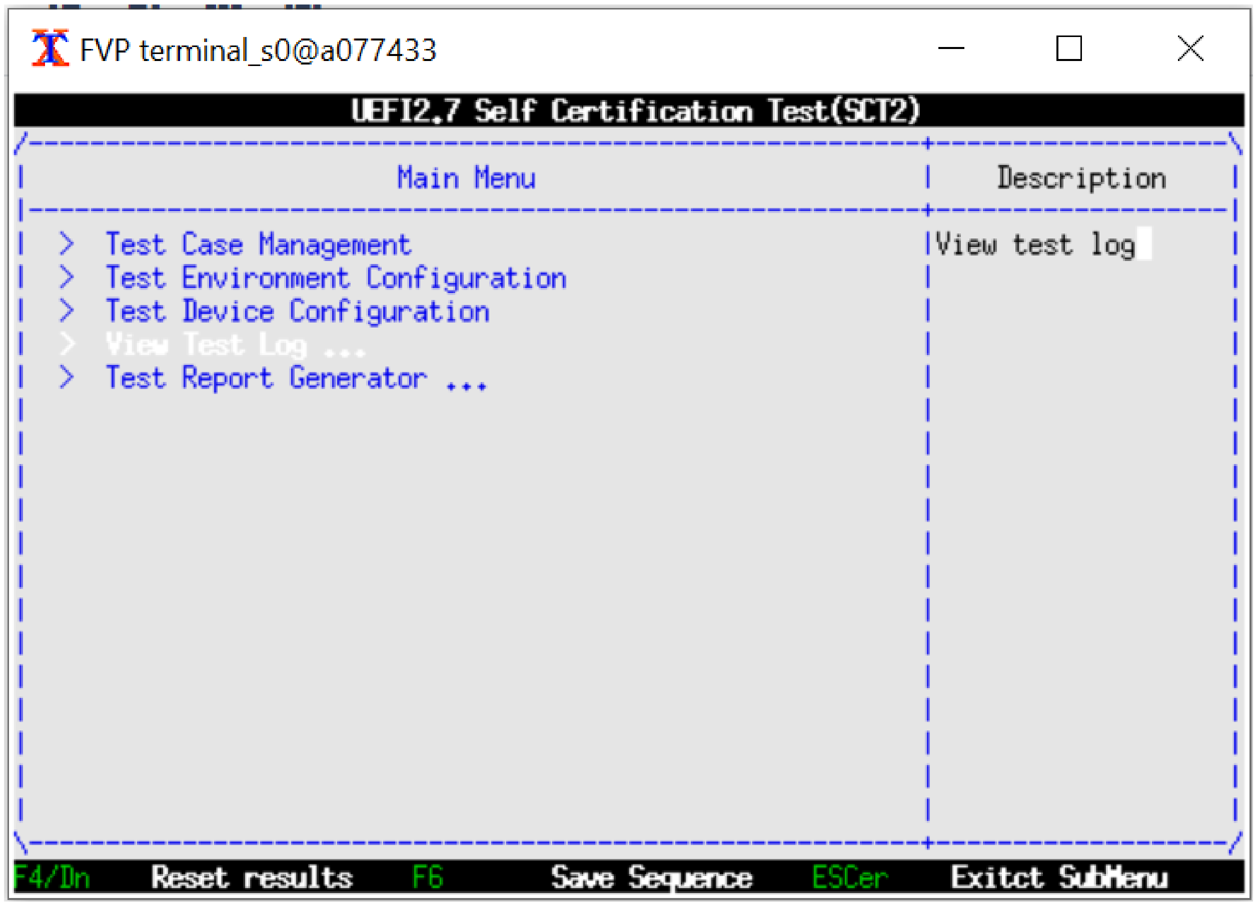
4- Select “Test Case Management”. Then, select any test you want to run. - To select a test tap “Space”, [1] should be printed in #Iter - To deselect a test tap again “Space”, [0] should be printed in #Iter



```
FVP terminal_s0@a077433
                    UEFI2.7 Self Certification Test(SCT2)
/--------------------------------------------------------+----------------\
|                     Main Menu                          |  Description   |
|--------------------------------------------------------+----------------|
|  > Test Case Management                                |Select and execute |
|  > Test Environment Configuration                      |test cases      |
|  > Test Device Configuration                           |                |
|  > View Test Log ...                                   |                |
|  > Test Report Generator ...                           |                |
\--------------------------------------------------------+----------------/
F4/Dn    Reset results    F6     Save Sequence    ESCer    Exitct SubMenu
```

5- Click on F9 to run the selected tests

6- Retrieve the test results in the “View Test Log…”



Copyright (c) 2020-2023, Arm Limited. All rights reserved.

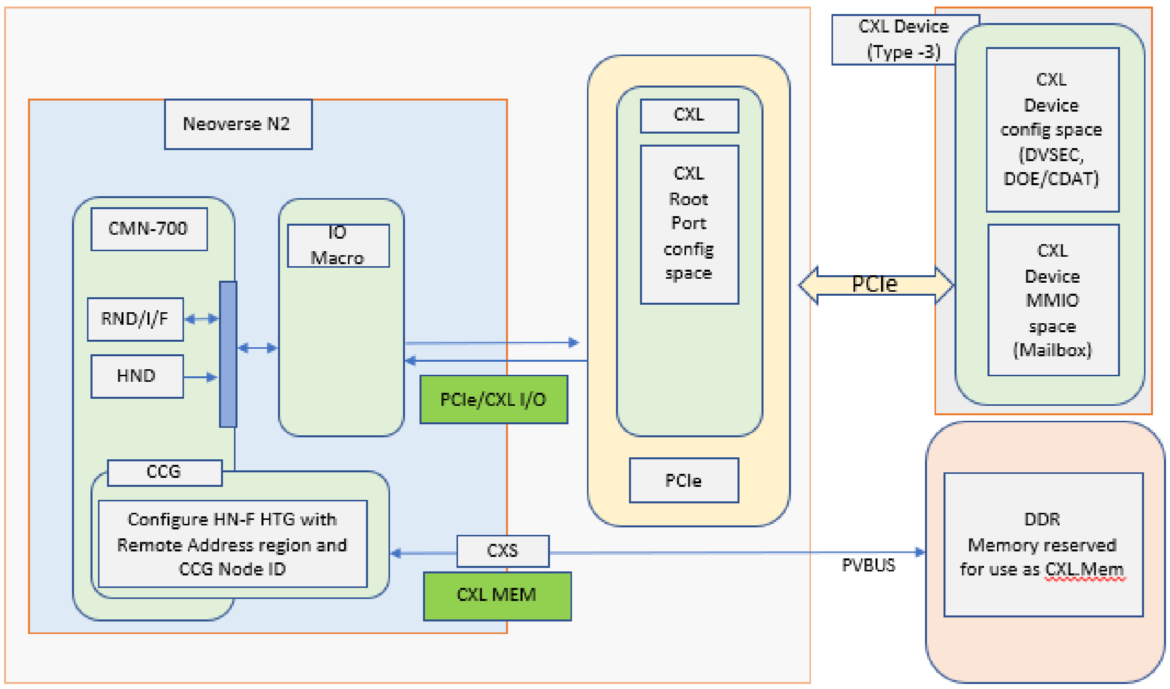
CHAPTER

FIVE

COMPUTE EXPRESS LINK (CXL)

5.1 Overview of CXL test

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.



Neoverse N2
CMN-700
RND/I/F
HND
IO Macro
CCG
Configure HN-F HTG with Remote Address region and CCG Node ID
CXS
CXL MEM
PCIe/CXL I/O
CXL
CXL Root Port config space
PCIe
PCIe
CXL Device (Type -3)
CXL Device config space (DVSEC, DOE/CDAT)
CXL Device MMIO space (Mailbox)
PVBUS
DDR Memory reserved for use as CXL.Mem

The above representation shows how CXL Type-3 device is modeled on Neoverse N2 refernce design paltform.

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.

5.2 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.

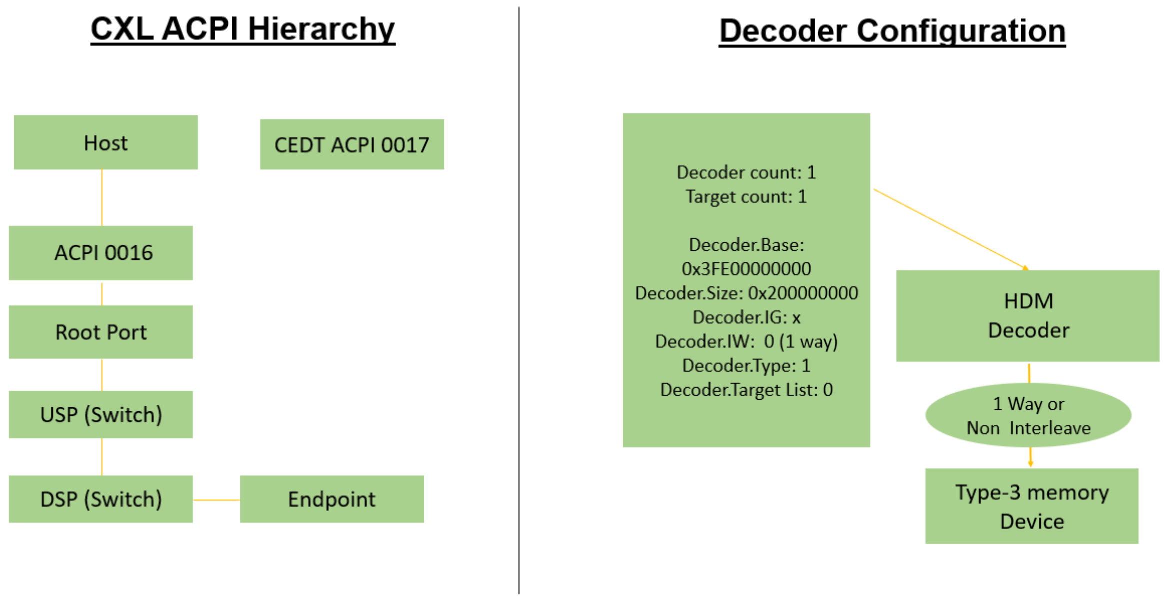
   Prepare HMAT table with required proximity, latency info.
5. The remote CXL memory will be represented to kernel as Memory only NUMA node.

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.

5.3 CXL with CEDT and Decoder configuation



5.4 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"

5.5 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
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.

5.6 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
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

CHAPTER

SIX

MEMORY SYSTEM RESOURCE PARTITIONING AND MONITORING (MPAM)

6.1 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.

6.2 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-2022.09.30-MPAM` tag for linux and repository 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.

6.3 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]
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
```

6.4 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 are as shown below.

```
/* MPAM_MSC_NODE 1 */
{
  RD_MPAM_MSC_NODE_INIT(0x1, 0x142601000, 0, MPAM_MSC_COUNT,
    RESOURCES_PER_MSC, FUNCTIONAL_DEPENDENCY_PER_RESOURCE)
},
/* MPAM_MSC_NODE 2 */
{
  RD_MPAM_MSC_NODE_INIT(0x2, 0x142641000, 0, MPAM_MSC_COUNT,
    RESOURCES_PER_MSC, FUNCTIONAL_DEPENDENCY_PER_RESOURCE)
},
```

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)
```

(continues on next page)

(continued from previous page)

```
      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.

6.5 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 RdN2Cd, 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);
~
```

```
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);
```

(continues on next page)

(continued from previous page)

```
    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();
    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;
```

(continues on next page)

(continued from previous page)

```
    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]
 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]
```

(continues on next page)

(continued from previous page)

```
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.

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

CHAPTER

SEVEN

POWER MANAGEMENT

7.1 ACPI Low Power Idle (LPI)

7.1.1 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.

7.1.2 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.

7.1.3 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.

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

7.2 Collaborative Processor Performance Control (CPPC)

7.2.1 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 functionaliry. 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.

7.2.2 Download and build the required platform software

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

7.2.3 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.

7.2.4 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-Fremont 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.

7.2.5 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.

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

7.3 Reboot and Shutdown support for RD-Fremont platform

7.3.1 Overview of the reboot modes supported

RD Fremont 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 Subsystem (RSS) and then the Manageability Control Processor (MCP). In response to the shutdown request, the RSS 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-Fremont 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 RSS. In response to the cold reboot request, MCP undertakes power-down activities and enter WFI mode. Meanwhile the RSS 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 RSS, 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:

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

7.3.2 Power-down sequence for RD-Fremont 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.

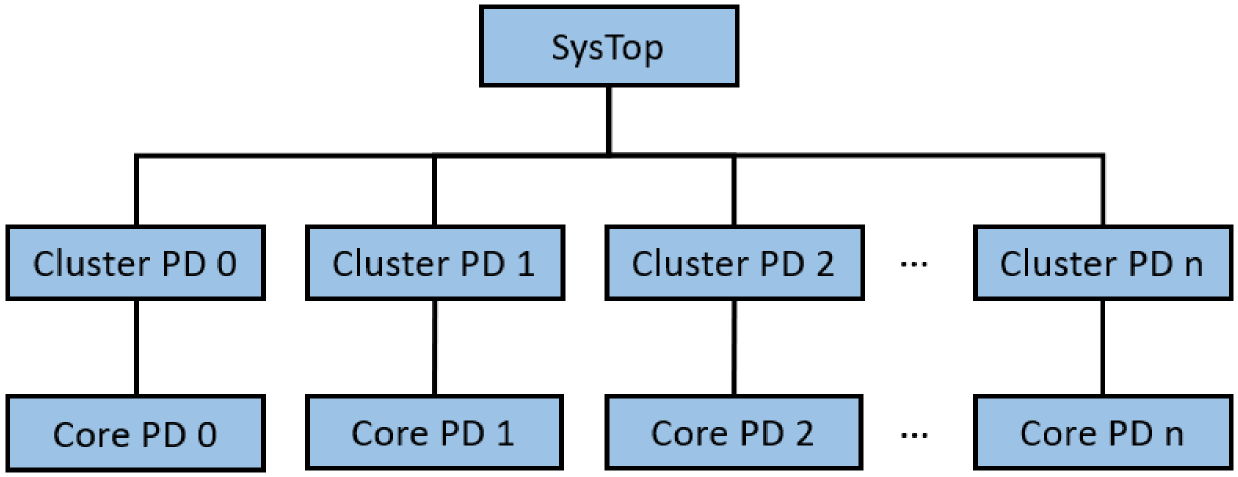
Subsequent to the switch to secure mode, CPU0 generates a Software Generated Interrupt (SGI) to bring the secondary CPUs into secure mode, triggering the execution of the ISR for the SGI. Within the ISR, secondary CPUs execute the power-down sequence, including disabling further interrupts and entering 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 Fremont 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 RSS using the same SCMI message format received from AP. This prompts RSS to execute its shutdown sequence. Subsequently, SCP notifies the Main Control Processor (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 RSS and MCP stages. In the case of a cold reboot, SCP alters the sequence by informing MCP before RSS.

Upon receiving the cold reboot message, MCP initiates its power-down sequence and enters WFI mode. RSS 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.

7.3.3 Download and build the required platform software

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

7.3.4 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
```

Copyright (c) 2024 Arm Limited. All rights reserved.

7.4 System Monitoring Control Framework (SMCF)

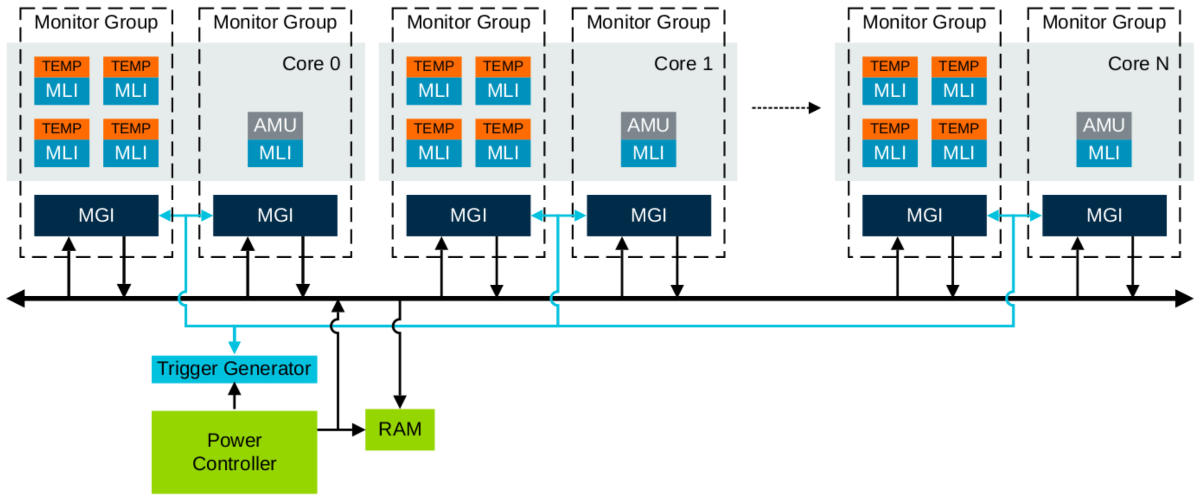
7.4.1 Overview of SMCF

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.

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:

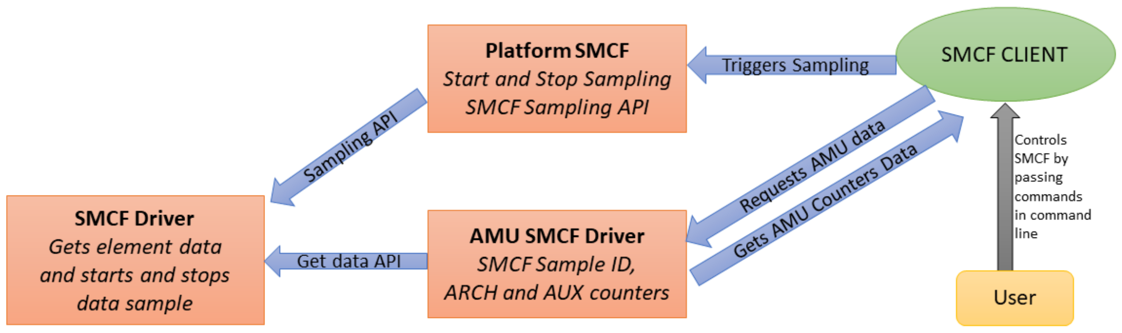
1. Manual Trigger : Initiated by the software for a single sample from the SMCF.
2. Periodic Sample: Software-driven continuous sampling at predefined interval.

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.

7.4.2 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 plaftform 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:



7.4.3 Download and build the required platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

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

7.4.4 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 RdFremontCfg1 platform 8 such instances will be there. An example output looks like below:

```
CLIENT: Data successfully fetched for MGI[0]
CLIENT: MGI[0] AMU_COUNTER[0] data = 3996611651
CLIENT: MGI[0] AMU_COUNTER[1] data = 137883084
CLIENT: MGI[0] AMU_COUNTER[2] data = 3996611651
CLIENT: MGI[0] AMU_COUNTER[3] data = 0
CLIENT: MGI[0] AMU_COUNTER[4] data = 0
CLIENT: MGI[0] AMU_COUNTER[5] data = 0
CLIENT: MGI[0] AMU_COUNTER[6] data = 0
```

(continues on next page)

(continued from previous page)

```
CLIENT: Data successfully fetched for MGI[1]
CLIENT: MGI[1] AMU_COUNTER[0] data = 16919141
CLIENT: MGI[1] AMU_COUNTER[1] data = 583696
CLIENT: MGI[1] AMU_COUNTER[2] data = 16919141
CLIENT: MGI[1] AMU_COUNTER[3] data = 0
CLIENT: MGI[1] AMU_COUNTER[4] data = 0
CLIENT: MGI[1] AMU_COUNTER[5] data = 0
CLIENT: MGI[1] AMU_COUNTER[6] data = 0
```

7.4.5 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.

Copyright (c) 2024, Arm Limited. All rights reserved.

**

CHAPTER

EIGHT

RELIABILITY, AVAILABILITY, AND SERVICEABILITY (RAS)

8.1 Reliability, Availability, and Serviceability (RAS)

8.1.1 Concept of RAS

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.

8.1.2 Overview

RAS spec divides the entire RAS architectural extension support into 2 into 2 categories

- ARMv8-A RAS Extension
- RAS system architecture

RAS architectural spec defines the hardware ras extensions the cpu and the system could implement to achieve desired level of RAS support. This document outlines concepts of RAS architecture important to understand the ras software architecture.

ARMv8-A RAS Extension define 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 define 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 for handling the given error e.g AP, SCP or MCP. This architectural definitions makes designing the software easier. Few component definitions that the RAS System architecture defines

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).

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).

8.1.3 Software Error Handling

There are couple of approaches to achieve error handling in software. They are

- Firmware First Error Handling.
- Kernel First Error Handling.

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 spec 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.

Here are example platform implementations for Firmware First Error Handling.

- *Base RAM ECC RAS support*
- *N2 CPU RAS support*

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 spec for RAS spec. 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.

Here is an example platform implementation for Kernel First Error Handling. *N2 CPU RAS support*

8.1.4 Error Injection Software

Error injeciton 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 design platform use the Error Injection (EINJ) ACPI table defined in the ACPI spec 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 platform firmare at EL3 implements the functionality to program the error injeciton registers. OSPM initiates the injection and generates an SPI interrupt to call in to platform 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. The steps to exercise EINJ feature can be found in *Base RAM ECC RAS support* and *N2 CPU RAS support*.

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

8.2 Base RAM ECC RAS Test

8.2.1 Overview

The Neoverse N2 reference design platform has support for both Secure and Non-Secure RAM. Secure RAM is defined in 0x0400_0000 - 0x04ff_ffff address range. And non-secure RAM is defined in 0x0600_0000 - 0x07ff_ffff address range. Both the regions are defined as part of Application Processor (AP) memory map. These regions are backed by ECC to support error detection and correction.

Here we use Secure RAM ECC feature to demonstrate 1-bit corrected error injection and handling. This test is based on Firmware First software error handling approach.

Note: This test is only supported on Neoverse N2 based reference design platforms.

8.2.2 1-bit CE error injection on Secure RAM

ECC module on Base RAM can detect and correct 1-bit CE that occur on RAM. Base element RAMs implements 6 register banks of RAS registers. One pair of register bank is assigned to AP, SCP and MCP components. One bank in each pair implements ECC RAS register for secure RAM and other implements register for non-secure RAM.

RAS register within each bank defines a ErrCtrl register. This register can be programmed to generate Corrected Error (CE) or Uncorrected Error (UE) errors in the RAM. After the ErrCtrl is programmed to inject desired error, injection software must initiate a read transaction to any Secure RAM location for injection to take effect. After that the error records will be populated with appropriate values and a corresponding fault or error interrupt will be generated.

Detailed Error injection software sequence is illustrated to inject 1-bit CE into Secure RAM.

- Map the AP Secure RAM ECC RAS registers device memory space.
- Program the ErrCtlr register to inject CE.
- **Program the ErrCtrl register to enable ECC support.**

  - mmio_write_32((AP_S_RAM_ECC_RAS_BASE + 0x004),INJECT_CE_BIT | ENABLE_ECC)

- **Read any Secure RAM region.**
  - data = *(volatile uint32_t *)SECURE_RAM_ADDR

8.2.3 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

8.2.4 Procedure to perform 1-bit CE injection and handling on Secure RAM

Boot upto Busybox

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

Secure RAM error handling test

Run below command to inject 1-bit CE on the Secure RAM. This test injects error to address location 0x0403_0500. This test used EINJ ACPI table to make the injection. Base element 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 Secure RAM errors.

```
echo 0x80020000 > /sys/kernel/debug/apei/einj/error_type
echo 1 > /sys/kernel/debug/apei/einj/error_inject
```

The firmware publishes this error to OSPM via standard error record format (CPER) for Memory errors. The kernel on reception of this error information logs this on the console.

```
{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: 0x0000000004030500
{1}[Hardware Error]:   physical_address_mask: 0x0000ffffffffffff
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

8.3 N2 CPU RAS Test

8.3.1 Overview

The Neoverse N2 core based reference design platform has support for 2 error nodes. The presence of these nodes thus enables RAS extension on Neoverse N2 core.

- Node 0: Includes the L3 memory system in the DSU.
- Node 1: Includes the private L1 and L2 memory systems in the core.

The RAM's in the N2 core support SED parity (Single Error Detect) and SECDED ECC (Single Error Correct Double Error Detect) capabilities.

Neoverse N2 core also supports inserting errors in the error detection logic to verify error handling software.

Note: The Neoverse N2 reference design platform is based on direct connect configuration and has no DSU. Hence Neoverse N2 reference design platform supports only one error node i.e Node1.

8.3.2 1-bit CE error injection on N2 CPU

Neoverse N2 core implements Pseudo Fault Generation registers. With the help of these register software can inject either CE, DE or UE into the cache RAMs.

Detailed Error injection software sequence is illustrated to inject 1-bit CE N2 CPU.

- **Select error record for L1 and L2 memory systems i.e. Node1**
  - write_errselr_el1 (1)
- **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 Corrected Error injection write**
  - write_cpu_pfg_ctrl_register (0xC0000040) // Generates FHI interrupt
- **Note the CE is generated when the CE counter implemented in the ErrMisc register overflows, so clear the cpu_pfg_ctrl register after the overflow happens to stop the injection.**
  - write_cpu_pfg_ctrl_register (0)

8.3.3 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

8.3.4 Select the Build option

N2 CPU supports both Firmware First and Kernel First Error handling. At given point of time either of the support can be enabled. Firmware First Support is enabled by default. To enable Kernel First support enable build option ARM_TF_RAS_KERNEL_FIRST and disable ARM_TF_RAS_FW_FIRST vice versa. Navigate to your workspace and

- **For Firmware First**
  - vim configs/rdn2cfg1/rdn2cfg1
  - Set ARM_TF_RAS_FW_FIRST = 1
  - Set ARM_TF_RAS_KERNEL_FIRST = 0
- **For Kernel First**
  - vim configs/rdn2cfg1/rdn2cfg1
  - Set ARM_TF_RAS_FW_FIRST = 0
  - Set ARM_TF_RAS_KERNEL_FIRST = 1

8.3.5 Procedure to perform 1-bit CE injection and handling on N2 CPU

Boot upto Busybox

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

N2 CPU error handling test

After the busybox boot is complete, use below commands to inject 1-bit CE on the N2 CPU. EINJ table debugfs enteries are used to inject the error. error_type field is set to 1 indicating its processor correctable error.

```
echo 1 > /sys/kernel/debug/apei/einj/error_type
echo 1 > /sys/kernel/debug/apei/einj/error_inject
```

Firmware First Error Handling

On successful error injection the firmware publishes this error to kernel via standard error record format (CPER) for Processor errors. The kernel on reception of this error information logs it on the console.

```
{1}[Hardware Error]: Hardware error from APEI Generic Hardware Error Source: 10
{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: ARM processor error
{1}[Hardware Error]:   MIDR: 0x00000000410fd490
{1}[Hardware Error]:   Multiprocessor Affinity Register (MPIDR): 0x0000000081000000
{1}[Hardware Error]:   running state: 0x1
{1}[Hardware Error]:   Power State Coordination Interface state: 0
{1}[Hardware Error]:   Error info structure 0:
{1}[Hardware Error]:   num errors: 135
{1}[Hardware Error]:    overflow occurred, error info is incomplete
{1}[Hardware Error]:    error_type: 1, TLB error
{1}[Hardware Error]:    error_info: 0x000000000402001f
{1}[Hardware Error]:     transaction type: Generic
{1}[Hardware Error]:     operation type: Generic error (type cannot be determined)
{1}[Hardware Error]:     TLB level: 0
{1}[Hardware Error]:     processor context not corrupted
{1}[Hardware Error]:     the error has been corrected
{1}[Hardware Error]:    physical fault address: 0x0000000000000000
```

Kernel First Error Handling

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. Logs from kernel first error handling test.

```
ERR1STATUS: 0x4e000012
ERR1MISC0: 0xe000000000
ERR1MISC1: 0x0
ERR1MISC2: 0x0
ERR1MISC3: 0x0
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

CHAPTER

NINE

VIRTUALIZATION TESTS

9.1 Virtualization using KVM

9.1.1 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.

9.1.2 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 hardare virtualization features on the platform and enable use of virtualized devices, such as console, net, and disk etc.

9.1.3 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.

9.1.4 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>" 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" 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" 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:

```
# 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.

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

9.2 KVM Unit Testing

9.2.1 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.

9.2.2 Build & Install

Download the software stack

Skip this section if the required sources have been downloaded and the host TAP interface has been setup.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page (including the setting up of the host TAP interface). Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Build the platform software

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 distro boot as given in *Distro Boot* guide,
- kvm-unit-tests built for kvmtool target,
- Kvmtool VMM.

Skip this section if a *Buildroot* or *Busybox* build is already performed for the platform software stack as the `kvmtool` and `kvm-unit-tests` are already built as part of the buildroot and busybox build.

- 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>
  ```

- A single script builds both `kvmtool` and `kvm-unit-tests` in the reference platform software stack. Run the below command to build the binaries:

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

  Supported command line options for above two commands are listed below:

  - <platform name>
    - Lookup for a platform name in *Platform Names*.
  - <command>

    - Supported commands are
        - `clean`
        - `build`
        - `package`

For example, to build the platform software stack and the kvmtool and kvm-unit-tests binaries for Rd-N2-Cfg1 platform run the following commands:

```
./build-scripts/build-test-uefi.sh -p rdn2cfg1 all
```

```
./build-scripts/build-kvmtool.sh -p rdn2cfg1 clean
./build-scripts/build-kvmtool.sh -p rdn2cfg1 build
./build-scripts/build-kvmtool.sh -p rdn2cfg1 package
```

Setup Satadisk Images

The kvm-unit-tests 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 v22.04 distro host OS.

9.2.3 Booting the platform for validation

- 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
```

Running Unit Testcases

After booting the host OS with a Linux distribution such as Ubuntu, create a workspace to begin with `kvm-unit-tests` example.

```
mkdir -p ~/unit-testing/
cd ~/unit-testing/
```

- Executing the unit testcase will require the `kvm-unit-tests` output directory and the `kvmtool` binary that are built as part of the reference software build as mentioned in the section *Build the platform software*. Copy these to host OS through network.

```
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 .
```

- Get the absolute path to the recently copied `lkvm` binary and run the below command to export the KVMTOOL as the target vmm to run kvm-unit-tests.

```
sudo su
export KVMTOOL=<Absolute path to kvmtool>/lkvm
```

- Finally, begin testing by executing the automated test script from `kvm-ut` directory that iterates over all available test implementation for the platform.

```
./run_tests.sh
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

9.3 Using non-discoverable devices connected to I/O virtualization block

9.3.1 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.

9.3.2 Download the platform software

Skip this section if the required sources have been downloaded.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page. Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

9.3.3 Build the platform software

Refer to the *Busybox Boot* page to build the reference design platform software stack and boot into busybox on the Neoverse RD FVP.

9.3.4 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 0x1080010000 32 0xabcd1234
```

- Similarly, test SRAM memory read by using the following example.

```
# devmem 0x1080010000 32
0xABCD1234
```

This completes the testing for non-PCI devices connected to the I/O virtualization block.

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

9.4 PCIe I/O virtualization

9.4.1 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.

9.4.2 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 extry 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" 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;
```

- 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" 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;
```

- 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/interrrupts
```

- 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.

```
# 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.
```

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

9.5 Virtual Interrupts And VGIC

9.5.1 Overview of Directly Injected vLPIs

Locality-specific Peripheral Interrupts (LPIs) are message based interupts 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 Transaltion Service (GIC-ITS) hardware provides such support by accepting a MMIO write and raing 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 everytime 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 transalation table and the vPE table alongwith redistributor's configuration and pending tables. With these addtions 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 resposible to trigger it to PE. This avoid any requiremnet of software (KVM) intrusion and makes it almost immediate trigger of vLPIs.

9.5.2 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.

9.5.3 Build & Install

Download the software stack

Skip this section if the required sources have been downloaded the and host TAP interface has been setup.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page (including the setting up of the host TAP interface). Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

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 afer 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
```

(continues on next page)

(continued from previous page)

```
GRUB_TIMEOUT=0
GRUB_CMDLINE_LINUX_DEFAULT="..."
#GRUB_TERMINAL=console

#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
```

9.5.4 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 successfull 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 unbind the smmute device from default smmute driver and attach it to `vfio-pci` driver on host. This is required to allow access of guest OS to this pci endpoint device. 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 "0000:08:00.1" | sudo tee /sys/bus/pci/devices/0000\:08\:00.1/driver/unbind
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
```

- 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; screen -r virt0;
```

- 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.
```

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

9.6 UEFI Based KVM Virtualization

9.6.1 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.

9.6.2 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.

9.6.3 Overview of ArmVirtKvmTool

ArmVirtKvmTool firmware is sepcifically 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 initalization by the firmware are parsed through flattened device tree (fdt) library. The firwmare 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.

9.6.4 Build & Install

Download the software stack

Skip this section if the required sources have been downloaded and the host TAP interface has been setup.

To obtain the required sources for the platform, follow the steps listed on the *Setup Workspace* page (including the setting up of the host TAP interface). Ensure that the platform software is downloaded before proceeding with the steps listed below. Also, note the host machine requirements listed on that page which is essential to build and execute the platform software stack.

Build the platform software

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
```

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.

9.6.5 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 & Install*. 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/rdn2/
  ↪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;
```

- 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.
```

Copyright (c) 2022-2023, Arm Limited. All rights reserved.

CHAPTER

TEN

RELEASE NOTES

10.1 RD-INFRA-2024.01.16

10.1.1 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.

10.1.2 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*

10.1.3 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.

10.1.4 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.
- RD-Fremont-Cfg2
  - Busybox boot, buildroot boot, virtual machine and kvm unit test in realm.

10.1.5 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

Copyright (c) 2024, Arm Limited. All rights reserved.

10.2 RD-INFRA-2023.12.22

10.2.1 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*

10.2.2 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.

10.2.3 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.3 RD-INFRA-2023.09.29

10.3.1 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

10.3.2 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.

10.3.3 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.4 RD-INFRA-2023.09.28

10.4.1 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

10.4.2 Known Limitations

- Hafnium is not enabled for RD-Fremont-Cfg2

10.4.3 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.

10.4.4 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.5 RD-INFRA-2023.06.30

10.5.1 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*

10.5.2 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.

10.5.3 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.6 RD-INFRA-2023.06.28

10.6.1 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

10.6.2 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.

10.6.3 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.

10.6.4 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.7 RD-INFRA-2023.03.31

10.7.1 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*

10.7.2 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.

10.7.3 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

Copyright (c) 2023, Arm Limited. All rights reserved.

10.8 RD-INFRA-2023.03.29

10.8.1 Release Description

- Introduce support for *RD-Fremont* and *RD-Fremont-Cfg1* platforms.

10.8.2 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.

10.8.3 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

Copyright (c) 2023, Arm Limited. All rights reserved.