Arm Kronos Reference Software Stack

Version v1.0

Arm Ltd.

May 03, 2024

CONTENTS

CHAPTER

ONE

OVERVIEW

1.1 Introduction

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

The **Arm Reference Design-1 AE**, or **RD-1 AE**, targets the Automotive segment and introduces the concept of a high-performance Arm® Neoverse™ V3AE Application Processor (Primary Compute) system augmented with an Arm® Cortex®-R82AE based Safety Island for scenarios where additional system safety monitoring is required. The system additionally includes a Runtime Security Engine (RSE) used for the secure boot of the system elements and the runtime Secure Services.

Throughout the following documentation, the alias "Kronos Reference Design" is used in place of Arm Reference Design-1 AE. For more information, including how to obtain the Technical Overview document, visit the Arm Reference Design-1 AE page on developer.arm.com.

A Fixed Virtual Platform (FVP) is available as part of the Reference Design. Further information on FVPs, including expected runtime performance and other capabilities, can be found at Arm Ecosystem FVPs.

This documentation covers the Kronos Reference Software Stack which together with the FVP allow for the exploration of baremetal and Xen hypervisor hosted Linux instances, Primary Compute to/from Safety Island communication mechanisms (for both baremetal and virtualized scenarios), and boot flows coordinated via a system root of trust. The Primary Compute firmware stack of Trusted Firmware-A, U-Boot, OP-TEE and Trusted Services is also aligned with the technologies and goals of the Arm SystemReady™ IR program.

1.2 Audience

The intended target audience of this document are software, hardware, and system engineers who are planning to evaluate and use the Arm Kronos Reference Software Stack.

It describes how to build and run images for the Arm Kronos Reference Design FVP (FVP_RD_Kronos) using the Yocto Project build framework. Basic instructions about the Yocto Project can be found in the Yocto Project Quick Start.

In addition to having Yocto related knowledge, the target audience also needs to have a certain understanding of the following technologies:

- Arm Firmware:
  - OP-TEE
  - Runtime Security Engine (RSE)
  - System Control Processor (SCP) Firmware

  - Local Control Processor (LCP) Firmware
  - Trusted Firmware-A (TF-A)
  - Trusted Services
- U-boot
- Xen Hypervisor
- Zephyr

1.2.1 Documentation Structure

- *User Guide*

  Provides guidance for configuring, building, and deploying the Reference Software Stack on the FVP and running and validating the supported functionalities.

- *Solution Design*

  Provides more advanced developer-focused details of the Reference Software Stack, its implementation, and dependencies.

- *License*

  Defines the license under which the Reference Software Stack is provided.

- *Release Notes*

  Documents new features, bug fixes, limitations, and any other changes provided under each Reference Software Stack release.

1.3 Reference Software Stack Overview

This Reference Software Stack is made available as part of the Arm Kronos Reference Design and is composed of multiple Open Source components which together form the proposed solution, including:

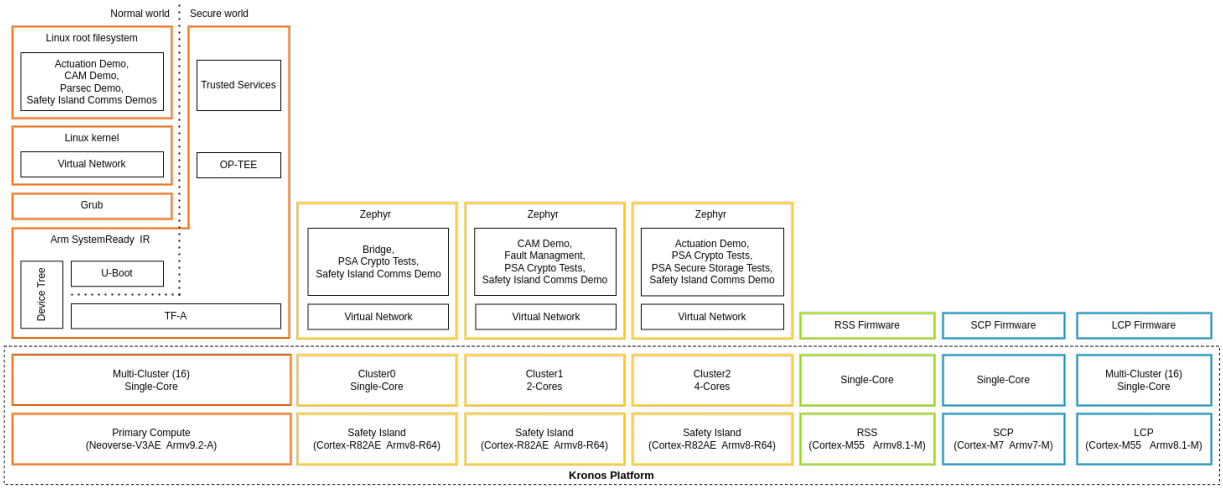
- The Runtime Security Engine (RSE) - referred to in this document as the Runtime Security Subsystem (RSS) - running an instance of Trusted Firmware-M, which offers boot, cryptography, and secure storage services.
- The Safety Island subsystem, running three instances of the Zephyr real-time operating system (RTOS).
- The firmware for the Primary Compute, using Trusted Firmware-A, U-Boot, OP-TEE and Trusted Services. These are configured to be aligned with Arm SystemReady IR.

The remaining software in the Primary Compute subsystem, based on the Cassini distribution, is available in two main architectures: baremetal and virtualization.

Baremetal Architecture

The Primary Compute boots a single rich operating system (real-time Linux with PREEMPT_RT patches).

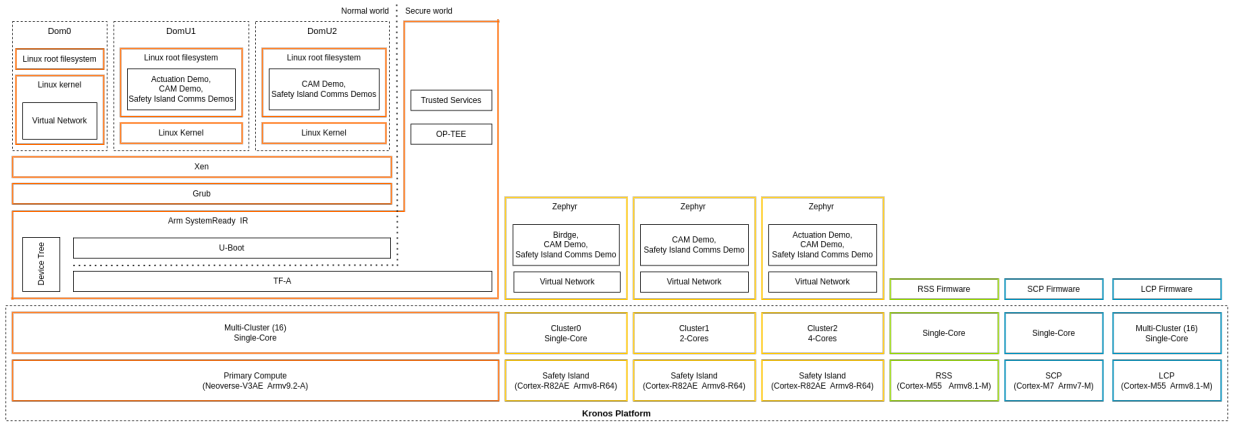
Arm Kronos Reference Software Stack High-Level Diagram - Baremetal Architecture



Virtualization Architecture

The Primary Compute boots into a type-1 hypervisor (Xen) using Arm's hardware virtualization support. There are three isolated, resource-managed virtual machines: Dom0 (privileged domain) and DomU1 and DomU2 (unprivileged domains).

Arm Kronos Reference Software Stack High-Level Diagram - Virtualization Architecture



1.4 Safety and Security Considerations

Kronos Reference Design software solutions are public example software projects that track and pull upstream components, incorporating their respective security fixes published over time. Arm partners are responsible for ensuring that the components they use contain all the required security fixes, if and when they deploy a product derived from Arm reference solutions.

1.5 Use-Cases

The Reference Software Stack demonstrates how the following features can be used to enhance the overall functional safety level of a high-performance compute platform:

- Critical Application Monitoring
- High reliability compute subsystem
- Safety Island Communication
- Transport Layer Security (TLS) with hardware cryptography support
- RSS Secure Services providing PSA Secure Storage and Crypto compliant APIs
- Arm SystemReady IR-aligned software stack
- Secure firmware update following Arm's Security Firmware Update Specification
- System Fault Handling for increased safety

The *Reproduce* section of the User Guide contains all the instructions necessary to fetch and build the source as well as to download the required FVP and launch the Use-Cases.

Following are the main Use-Cases implemented by the Reference Software Stack.

1.5.1 Critical Application Monitoring Demo

Critical Application Monitoring (CAM) is a project that implements a solution for monitoring critical applications using a service running on a higher safety level system. This demo deploys CAM components on the Kronos FVP to demonstrate the feasibility of the Safety Island monitoring solution.

Refer to *Critical Application Monitoring Demo* for more information.

1.5.2 Safety Island Actuation Demo

The Safety Island Actuation demo consists of the Arm SystemReady IR-aligned firmware along with Linux-based software on the Primary Compute and Zephyr application on the Safety Island to demonstrate automotive workloads. Refer to *Safety Island Actuation Demo* for more information.

1.5.3 Safety Island Communication Demo

The Safety Island Communication demo demonstrates via HIPC (Heterogeneous Inter-processor Communication), the networking between:

- Primary Compute and the three Safety Island clusters.
- Safety Island clusters.

Refer to *Heterogeneous Inter-Processor Communication (HIPC)* for more information on HIPC.

1.5.4 Parsec-enabled TLS Demo

The Parsec-enabled TLS demo illustrates a HTTPS session where a Transport Layer Security (TLS) connection is established, and a simple webpage is transferred. The TLS session consists of both symmetric and asymmetric cryptographic operations. The symmetric operations are executed by Mbed TLS in Linux userspace on the Primary Compute. The asymmetric operations are carried out by Parsec. While the backend of the Parsec service is based on RSS cryptographic runtime service. Refer to *Parsec-enabled TLS Demo* for more information.

1.5.5 Primary Compute PSA Protected Storage and Crypto APIs Architecture Test Suite

The PSA Protected Storage and Crypto architecture test suites are a set of examples of the invariant behaviors that are specified in the PSA Protected Storage APIs and PSA Crypto APIs specifications respectively.

Both suites are used to verify whether these behaviors are implemented correctly in our system. This suites contain self-checking and portable C-based tests with directed stimulus.

Refer to *Primary Compute Secure Services* for more information.

1.5.6 Safety Island PSA Secure Storage APIs Architecture Test Suite

The PSA Secure Storage architecture test suite is a set of examples of the invariant behaviors that are specified in the PSA Secure Storage APIs specification.

This suite is used to verify whether these behaviors are implemented correctly in our system. This suite contains self-checking and portable C-based tests with directed stimulus.

Refer to *PSA Secure Storage APIs Architecture Test Suite* for more information.

1.5.7 Safety Island PSA Crypto APIs Architecture Test Suite

The PSA Crypto architecture test suite is a set of examples of the invariant behaviors that are specified in the PSA Crypto APIs specification.

This suite is used to verify whether the PSA Crypto APIs provided on Safety Island are correctly implemented.

Refer to *PSA Crypto APIs Architecture Test Suite* for more information.

1.5.8 Fault Management Demo

The Fault Management subsystem for the Safety Island demonstrates the injection, reporting and collation of faults from supported hardware to support the design of safety-critical systems.

Refer to *Fault Management* for more information.

1.5.9 Arm SystemReady IR Validation

Arm SystemReady is a compliance certification program based on a set of hardware and firmware standards that enable interoperability with generic off-the-shelf operating systems and hypervisors. Refer to *Arm SystemReady IR* for more information.

1.5.10 Linux Distribution Installation

Demonstrates the installation of two unmodified generic UEFI distribution images, Debian and openSUSE, fulfilling Arm SystemReady requirements.

1.5.11 Secure Firmware Update

Demonstrates an implementation of Secure Firmware Update initiated from the Primary Compute and follows the Platform Security Firmware Update Specification. Refer to *Secure Firmware Update* for more information.

1.6 Repository Structure

The `kronos` repository (https://gitlab.arm.com/automotive-and-industrial/kronos-ref-stack/kronos) is structured as follows:

- `kronos`:
  - `yocto`

    Directory implementing the `meta-kronos` Yocto layer as well as kas build configuration files.
  - `components`

    Directory containing source code for components which can either be used directly or as part of the `meta-kronos` Yocto layer.
  - `documentation`

    Directory which contains the documentation sources, defined in ReStructuredText (`.rst`) format for building via `sphinx`.

1.7 Repository License

The repository's standard license is the MIT license (more details in *License*), under which most of the repository's content is provided. Exceptions to this standard license relate to files that represent modifications to externally licensed works (for example, patch files). These files may therefore be included in the repository under alternative licenses in order to be compliant with the licensing requirements of the associated external works.

1.8 Contributions and Issue Reporting

This project has not put in place a process for contributions currently.

To report issues with the repository such as potential bugs, security concerns, or feature requests, submit an Issue via GitLab Issues, following the project's template.

1.9 Feedback and Support

To request support contact Arm at support@arm.com. Arm licensees may also contact Arm via their partner managers.

CHAPTER

TWO

USER GUIDE

2.1 Reproduce

This section of the User Guide describes how to download, configure, build and execute this Reference Software Stack.

2.1.1 Introduction

This Reference Software Stack uses the kas menu tool to configure and customize the different *Use-Cases* via a set of configuration options provided in the configuration menu.

Note: All command examples on this page from the HTML document format can be copied by clicking the copy button. In the PDF document format, be aware that special characters are added when lines get wrapped.

2.1.2 Build Host Environment Setup

System Requirements

- x86_64 or aarch64 host to build the stack and execute the Kronos FVP
- Ubuntu Desktop or Server 20.04 Linux distribution
- At least 500GiB of free disk for the download and builds
- At least 32GiB of RAM memory
- At least 12GiB of swap memory

Install Dependencies

- Follow the Yocto Project documentation on how to install the essential packages required for the build host. The packages needed to build the Yocto Project documentation manuals are not required.
- Install the kas tool and its optional dependency (to use the "menu" plugin):

```
sudo -H pip3 install --upgrade kas==4.2 && sudo apt install python3-newt
```

  For more details on kas installation, see kas Dependencies & installation.

- Install tmux (required for the `runfvp` tool):

```
sudo apt install tmux
```

2.1.3 Download

Note: Performing the builds and FVP execution in **a tmux session is mandatory** for Kronos because the `runfvp` tool that invokes the Kronos FVP expects the presence of a tmux session to attach its spawned tmux windows for console access to the processing elements. Refer to Tmux Documentation for more information on the usage of tmux. It is recommended to change the default `history-limit` by adding `set-option -g history-limit 3000` to `~/.tmux.conf` before starting tmux.

Start a new tmux session, via:

```
tmux new-session -s kronos
```

To reconnect to an existing tmux session:

```
tmux attach -t kronos
```

Download the `kronos` repository using Git and checkout on the kronos branch, via:

```
mkdir -p ~/kronos
cd ~/kronos
git clone https://git.gitlab.arm.com/automotive-and-industrial/kronos-ref-stack/kronos.
↪git --branch v1.0
```

2.1.4 Reproducing the Use-Cases

General

Kas Build

The Kronos stack has a kas configuration menu that can be used to build the *Use-Cases*. It can also apply customizable parameters to build different Reference Software Stack Architecture types.

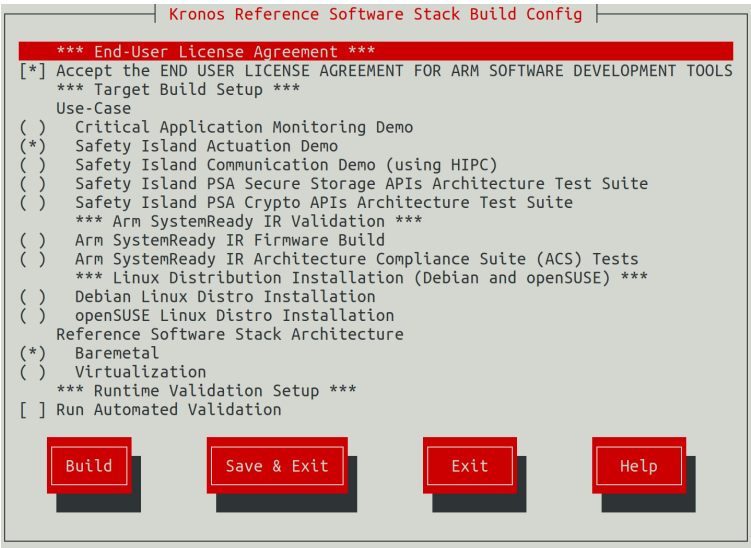
Note: Before running the configuration menu, ensure it is done inside a tmux session.

To run the configuration menu:

```
kas menu kronos/Kconfig
```

Note: To build and run any image for the Kronos FVP the user has to accept its EULA, which can be done by selecting the corresponding configuration option in the build setup. The Safety Island Actuation Demo is built as part of the default deployment.

The kas build configuration menu selections performed in each use-case are saved. Ensure to only select the options mentioned in the use-case reproduce steps and deselect any other non-relevant ones.



FVP

The `runfvp` tool that invokes the Kronos FVP creates one tmux terminal window per processing element. The default window displayed will be that of the Primary Compute terminal titled as `terminal_ns_uart0`. User may press `Ctrl-b w` to see the list of tmux windows and use arrow keys to navigate through the windows and press `Enter` to select any processing element terminal.

The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal.

Note: FVPs, and Fast Models in general, are functionally accurate, meaning that they fully execute all instructions correctly, however they are not cycle accurate. The main goal of the Reference Software Stack is to prove functionality only, and should not be used for performance analysis.

Critical Application Monitoring Demo

The demo can be run on the Baremetal Architecture or Virtualization Architecture. See *Critical Application Monitoring Demo* for further details.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Critical Application Monitoring Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

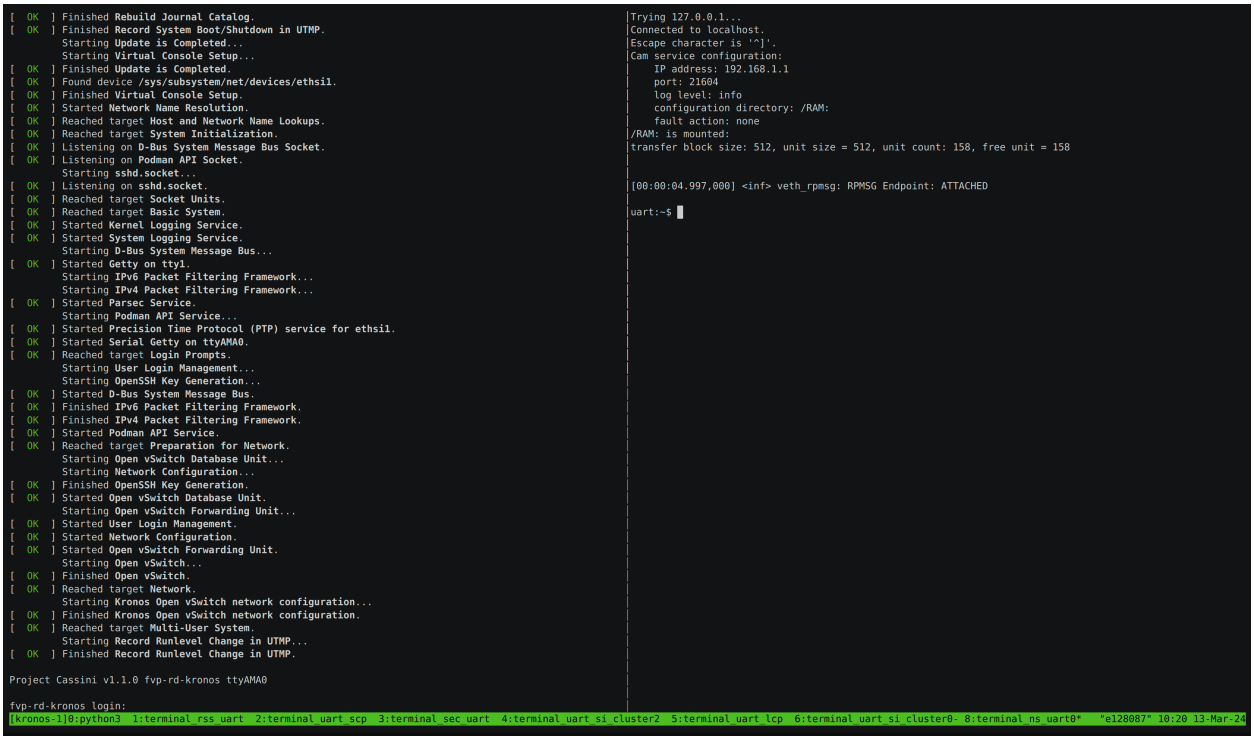
The user should wait for the system to boot and for the Linux prompt to appear.

The Safety Island (SI) Cluster 1 terminal running `cam-service` is available via the tmux window titled `terminal_uart_si_cluster1`. For ease of navigation, it's recommended to join the SI Cluster 1 terminal window to the Primary Compute terminal window in order to issue commands on it.

Follow the steps below to achieve the same:

1. Ensure that the tmux window titled `terminal_ns_uart0` is selected. If not, press `Ctrl-b w` from the tmux session, navigate to the tmux window titled `terminal_ns_uart0` using the arrow keys, then press the `Enter` key.
2. Press `Ctrl-b :` and then type `join-pane -s :terminal_uart_si_cluster1 -h` followed by pressing `Enter` key to join the SI Cluster 1 terminal window to the Primary Compute terminal window.

Refer to the following image of the tmux panes rearrangement. Panes can be navigated using `Ctrl-b` followed by the arrow keys.



The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Run the Demo

Before running `cam-app-example`, `.csd` files corresponding to event streams produced by `cam-app-example` must be created and deployed to the system where `cam-service` runs (in this case SI Cluster 1). Run `cam-app-example` in calibration mode and then use `cam-tool` to generate the `.csd` files.

1. Start `cam-app-example` in calibration mode from the Primary Compute terminal:

   ```
   cam-app-example -u 11085ddc-bc10-11ed-9a44-7ef9696e -t 3000 -c 10 -s 4 -C
   ```

   The stream event log files (`.csel`) for each stream are generated. The output should look as below:

   ```
   Cam application configuration:
       Service IP address: 127.0.0.1
       Service port: 21604
       UUID base: 11085ddc-bc10-11ed-9a44-7ef9696e
       Stream count: 4
       Processing period (ms): 3000
       Processing count: 10
       Multiple connection support: false
       Calibration mode support: true
       Calibration directory: ./[uuid].csel
       Fault injection support: false
       Event(s) interval time (ms): 0
   Using libcam v1.0
   Starting activity...
   Starting activity...
   Starting activity...
   Starting activity...
       Stream 0 sends event 0
       Stream 1 sends event 0
       Stream 2 sends event 0
       Stream 3 sends event 0
       ...
   ```

   List the files generated:

   ```
   ls -1 *.csel
   ```

   The stream event log files can be shown as below:

   ```
   11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
   11085ddc-bc10-11ed-9a44-7ef9696e0001.csel
   11085ddc-bc10-11ed-9a44-7ef9696e0002.csel
   11085ddc-bc10-11ed-9a44-7ef9696e0003.csel
   ```

2. Run `cam-tool` from the Primary Compute terminal to analyze stream event log files and convert them to stream configuration files (`.csc.yml`).


```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
```

The analysis result is reported from the Primary Compute terminal as below, the timeout value might change:

```
CAM event log analyze report:
Input event log file:                   11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
Output configuration file:              analyzed.csc.yml
Stream UUID:                            11085ddc-bc10-11ed-9a44-7ef9696e0000
Stream name:                            CAM STREAM  0
Timeout between init and start:         300000
Timeout between start and event:        450000
Application running times:              1
Processing count in each run:           [10]

Event ID        timeout
0               4000106
```

The stream configuration files contain human-readable settings used for the deployment phase of a critical application. Users can modify this configuration, for example to fine tune timeout values depending on the system capabilities.

Run `cam-tool` three more times for each of the other three streams.

```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csel
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csel
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csel
```

Then, use the `cam-tool pack` command for each of the streams to generate deployment data.

```
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csc.yml
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csc.yml
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csc.yml
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csc.yml
```

3. Run the `cam-tool deploy` command from the Primary Compute terminal to transfer the generated stream deployment data to SI Cluster 1 (where `cam-service` is running):

```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

After that, the stream data of `11085ddc-bc10-11ed-9a44-7ef9696e0000` is deployed to the `cam-service` file system.

Running `cam-tool deploy` three more times can deploy the data of three other streams to `cam-service`.

```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csd -a 192.168.1.1
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csd -a 192.168.1.1
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal after each one of the `cam-tool deploy` command should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

List all the files from the SI Cluster 1 terminal:

```
fs ls RAM:/
```

The stream deployment data can be shown as below:

```
11085ddc-bc10-11ed-9a44-7ef9696e0000.csd
11085ddc-bc10-11ed-9a44-7ef9696e0001.csd
11085ddc-bc10-11ed-9a44-7ef9696e0002.csd
11085ddc-bc10-11ed-9a44-7ef9696e0003.csd
```

4. Start `cam-app-example` from the Primary Compute terminal to create an application with four streams. Each stream sends an event message 10 times with a period of 3000 milliseconds.

```
cam-app-example -u 11085ddc-bc10-11ed-9a44-7ef9696e -t 3000 -c 10 -s 4 -a 192.168.1.
↪1
```

The following configure messages are expected from the Primary Compute terminal:

```
Cam application configuration:
    Service IP address: 192.168.1.1
    Service port: 21604
    UUID base: 11085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 4
    Processing period (ms): 3000
    Processing count: 10
    Multiple connection support: false
    Calibration mode support: false
    Fault injection support: false
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
Starting activity...
Starting activity...
```

And the log of sent event messages are shown repeatedly:

```
Stream 0 sends event 0
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
Stream 0 sends event 0
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
...
```


As observed from the SI Cluster 1 terminal, `cam-service` is loading four stream deployment files for monitoring. In the following log, the stream messages are received and processed by it:

```
Connection 4 is created.
Init Message
Stream 11085ddc-bc10-11ed-9a44-7ef9696e0001 configuration is loaded.
Init Message
Stream 11085ddc-bc10-11ed-9a44-7ef9696e0000 configuration is loaded.
Init Message
Stream 11085ddc-bc10-11ed-9a44-7ef9696e0002 configuration is loaded.
Init Message
Stream 11085ddc-bc10-11ed-9a44-7ef9696e0003 configuration is loaded.
Start Message
Start Message
Start Message
Start Message
Event Message
Event Message
Event Message
Event Message
Event Message
# Repeated event messages
...
```

5. `cam-app-example` has a mode to inject a fault to test the CAM framework. Run `cam-app-example` again from the Primary Compute terminal with fault injection to event stream 0:

```
cam-app-example -u 11085ddc-bc10-11ed-9a44-7ef9696e -t 3000 -c 10 -s 4 -f -S 0 -T␣
↪1000 -a 192.168.1.1
```

The following configure messages are expected from the Primary Compute terminal:

```
Cam application configuration:
    Service IP address: 192.168.1.1
    Service port: 21604
    UUID base: 111085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 4
    Processing period (ms): 3000
    Processing count: 10
    Multiple connection support: false
    Calibration mode support: false
    Fault injection support: true
    Fault injection time: 1000
    Fault injection stream: 0
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
Starting activity...
Starting activity...
```

And the log of sent event messages are shown repeatedly:

```
Stream 0 sends event 0
```

(continues on next page)

(continued from previous page)

```
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
Stream 0 sends event 0
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
...
```

The fault happens 1000ms after stream initialization. At that time `cam-service` should detect a stream temporal error with the following output from the SI Cluster 1 terminal.

```
# Repeated event messages
...
ERROR: Stream temporal error:
ERROR:     stream_name: CAM STREAM  0
ERROR:     stream_uuid: 11085ddc-bc10-11ed-9a44-7ef9696e0000
ERROR:     event_id: 0
ERROR:     time_received: 0
ERROR:     time_expected: 1710328901375511
# Repeated event messages
...
ERROR: Stream state error:
ERROR:     stream_name: CAM STREAM  0
ERROR:     stream_uuid: 11085ddc-bc10-11ed-9a44-7ef9696e0000
ERROR:     timestamp: 1710328927375278
ERROR:     current_state: Failed state
ERROR:     requested_state: In-progress state
```

Note: `time_received: 0` should be ignored as the `time_received` is not set during a fault.

6. To shut down the FVP and terminate the emulation automatically, issue the following command on the Primary Compute terminal.

```
shutdown now
```

The below messages indicate the shutdown process is complete.

```
[  OK  ] Finished System Power Off.
[  OK  ] Reached target System Power Off.
reboot: Power down
```


Automated Validation

Ensure the creation of the initial firmware flash images because previously updated firmware will lead to failure of the tests.

```
kas shell -c "bitbake ap-flash-image rss-flash-image -C image"
```

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Critical Application Monitoring Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 40 minutes to complete. See *Integration Tests Validating the Critical Application Monitoring Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_40_cam.CAMServiceTest.test_cam_service_boot_on_si: PASSED (0.00s)
RESULTS - test_40_cam.CAMTest.test_cam_app_example_to_service_on_si: PASSED (36.89s)
RESULTS - test_40_cam.CAMTest.test_cam_app_example_to_service_on_si_with_multiple_
↪connections: PASSED (36.98s)
RESULTS - test_40_cam.CAMTest.test_cam_ptp_sync: PASSED (0.00s)
RESULTS - test_40_cam.CAMTest.test_cam_tool_deploy_to_si: PASSED (75.69s)
RESULTS - test_40_cam.CAMTest.test_cam_tool_pack: PASSED (74.72s)
RESULTS - test_40_cam.CAMTest.test_data_calibration: PASSED (187.73s)
RESULTS - test_40_cam.CAMTest.test_logical_check_on_si: PASSED (15.63s)
RESULTS - test_40_cam.CAMTest.test_temporal_check_on_si: PASSED (44.32s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Virtualization Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Virtualization Architecture image:

1. Select `Critical Application Monitoring Demo` from the `Use-Case` menu.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The user should wait for the system to boot and for the Linux prompt to appear.

The Safety Island (SI) Cluster 1 terminal running `cam-service` is available via the tmux window titled `terminal_uart_si_cluster1`. For ease of navigation, it's recommended to join the SI Cluster 1 terminal window to the Primary Compute terminal window in order to issue commands on it.

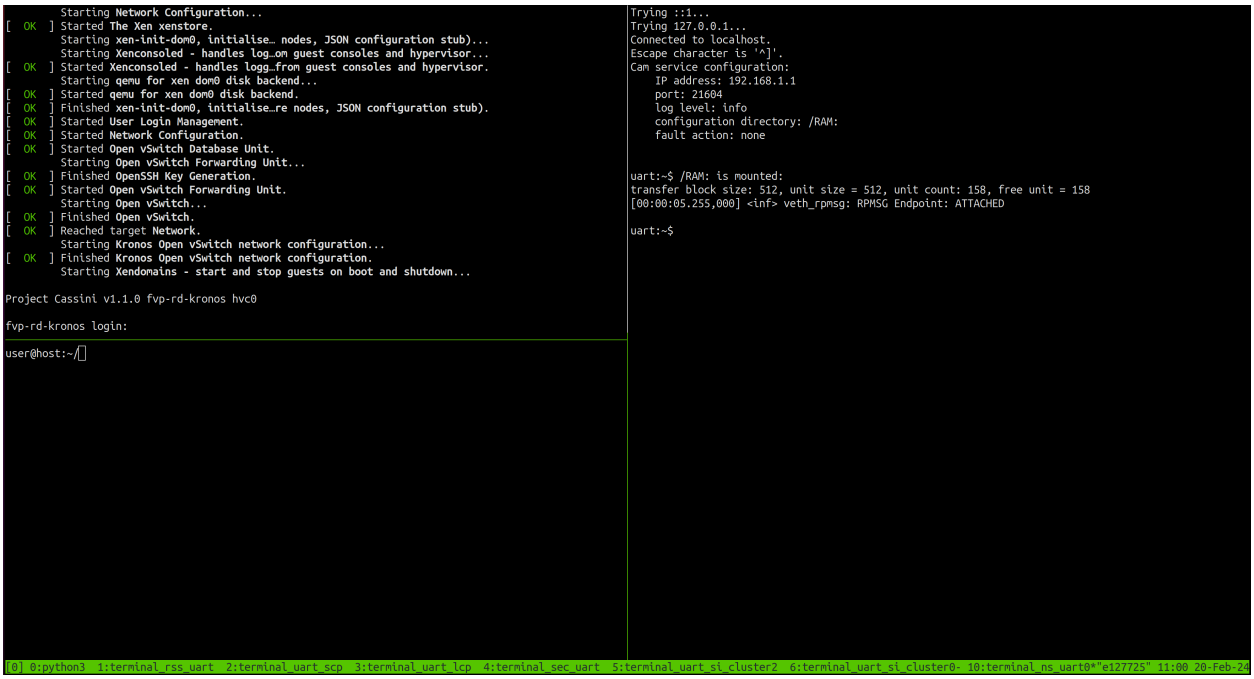
Follow the steps below to achieve the same:

1. Ensure that the tmux window titled `terminal_ns_uart0` is selected. If not, press `Ctrl-b w` from the tmux session, navigate to the tmux window titled `terminal_ns_uart0` using the arrow keys, then press the `Enter` key.
2. Press `Ctrl-b :` and then type `join-pane -s :terminal_uart_si_cluster1 -h` followed by pressing `Enter` key to join the SI Cluster 1 terminal window to the Primary Compute terminal window.

Since both DomU1 and DomU2 will be used to run `cam-app-example`, it is also recommended to create a tmux pane to connect to DomU2.

1. Press `Ctrl-b` and the arrow keys to navigate to the `terminal_ns_uart0` pane.
2. Press `Ctrl-b "` to split the pane horizontally. The bottom pane will be used to connect to DomU2.

Please refer to the following image of the tmux panes rearrangement. Panes can be navigated using `Ctrl-b` followed by the arrow keys.



The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password

in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Note: A message similar to the following might appear in the Primary Compute terminal `(XEN) d2v0: vGICR: SGI: unhandled word write 0x000000ffffffff to ICACTIVER0`, this is an expected behavior.

Run the Demo

Before running `cam-app-example`, `.csd` files corresponding to event streams produced by `cam-app-example` must be created and deployed to the system where `cam-service` runs (in this case SI Cluster 1). Run `cam-app-example` in calibration mode and then use `cam-tool` to generate the `.csd` files.

1. From the Primary Compute terminal, enter the DomU1 console using the `xl` tool:

   ```
   xl console domu1
   ```

   DomU1 can be logged into as `root` user without a password in the Linux terminal. This command will provide a console on the DomU1. To exit, enter `Ctrl-]` (to access the FVP telnet shell), followed by typing `send esc` into the telnet shell and pressing `Enter`. See the xl documentation for further details.

2. To improve the readability of commands and output on the DomU1 console, run the command below:

   ```
   stty rows 76 cols 282
   ```

3. From the host terminal, SSH to the FVP then enter the DomU2 console using the `xl` tool:

   ```
   ssh root@127.0.0.1 -p 2222
   xl console domu2
   ```

   DomU2 can be logged into as `root` user without a password in the Linux terminal. This command will provide a console on the DomU2. To exit, enter `Ctrl-]` (to access the FVP telnet shell), followed by typing `send esc` into the telnet shell and pressing `Enter`. See the xl documentation for further details.

4. To improve the readability of commands and output on the DomU2 console, run the command below:

   ```
   stty rows 76 cols 282
   ```

5. From the DomU1 terminal, check that the clock is synchronized using the command `timedatectl`, one of the line of its output needs to be `System clock synchronized: yes` to confirm that the clock is synchronized:

   ```
   timedatectl
   ```

   The output should look as below, the date and time can differ, in case the `System clock synchronized:` shows the `no` value, allow at least 1 minute for the system to settle and for the clock to synchronize, afterwards repeat the step 5 until `System clock synchronized: yes` is shown in the output:

   ```
                  Local time: Thu 2024-03-14 12:56:26 UTC
              Universal time: Thu 2024-03-14 12:56:26 UTC
                    RTC time: n/a
                   Time zone: UTC (UTC, +0000)
   System clock synchronized: yes
   ```

(continues on next page)

(continued from previous page)

```
             NTP service: n/a
         RTC in local TZ: no
```

6. Start `cam-app-example` in calibration mode from the DomU1 terminal:

```
cam-app-example -u 11085ddc-bc10-11ed-9a44-7ef9696e -t 3000 -c 10 -s 4 -C
```

The stream event log files (`.csel`) for each stream are generated. The output should look as below:

```
Cam application configuration:
    Service IP address: 127.0.0.1
    Service port: 21604
    UUID base: 11085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 4
    Processing period (ms): 3000
    Processing count: 10
    Multiple connection support: false
    Calibration mode support: true
    Calibration directory: ./[uuid].csel
    Fault injection support: false
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
Starting activity...
Starting activity...
    Stream 0 sends event 0
    Stream 1 sends event 0
    Stream 2 sends event 0
    Stream 3 sends event 0
    ...
```

List the files generated:

```
ls -1 *.csel
```

The stream event log files can be shown as below:

```
11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
11085ddc-bc10-11ed-9a44-7ef9696e0001.csel
11085ddc-bc10-11ed-9a44-7ef9696e0002.csel
11085ddc-bc10-11ed-9a44-7ef9696e0003.csel
```

7. Run `cam-tool` from the DomU1 terminal to analyze stream event log files and convert them to stream configuration files (`.csc.yml`).

```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
```

The analysis result is reported from the DomU1 terminal as below, the timeout value might change:

```
CAM event log analyze report:
Input event log file:                   11085ddc-bc10-11ed-9a44-7ef9696e0000.csel
Output configuration file:              analyzed.csc.yml
```

(continues on next page)

(continued from previous page)

```
Stream UUID:                              11085ddc-bc10-11ed-9a44-7ef9696e0000
Stream name:                              CAM STREAM  0
Timeout between init and start:           300000
Timeout between start and event:          450000
Application running times:                1
Processing count in each run:             [10]

Event ID        timeout
0               4000072
```

The stream configuration files contain human-readable settings used for the deployment phase of a critical application. Users can modify this configuration, for example to fine tune timeout values depending on the system capabilities.

Run `cam-tool` three more times for each of the other three streams.

```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csel
```

```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csel
```

```
cam-tool analyze -m 1000000 -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csel
```

Then, use the `cam-tool pack` command for each of the streams to generate deployment data.

```
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csc.yml
```

```
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csc.yml
```

```
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csc.yml
```

```
cam-tool pack -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csc.yml
```

8. Run the `cam-tool deploy` command from the DomU1 terminal to transfer the generated stream deployment data to SI Cluster 1 (where `cam-service` is running):

```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0000.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

After that, the stream data of `11085ddc-bc10-11ed-9a44-7ef9696e0000` is deployed to the `cam-service` file system.

Running `cam-tool deploy` three more times can deploy the data of three other streams to `cam-service`.

```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0001.csd -a 192.168.1.1
```

```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0002.csd -a 192.168.1.1
```



```
cam-tool deploy -i 11085ddc-bc10-11ed-9a44-7ef9696e0003.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal after each one of the `cam-tool deploy` command should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

9. From the DomU2 terminal, check that the clock is synchronized using the command `timedatectl`, one of the line of its output needs to be `System clock synchronized: yes` to confirm that the clock is synchronized:

```
timedatectl
```

The output should look as below, the date and time can differ, in case the `System clock synchronized:` shows the `no` value, allow at least 1 minute for the system to settle and for the clock to synchronize, afterwards repeat the step 9 until `System clock synchronized: yes` is shown in the output:

```
               Local time: Thu 2024-03-14 12:56:26 UTC
           Universal time: Thu 2024-03-14 12:56:26 UTC
                 RTC time: n/a
                Time zone: UTC (UTC, +0000)
System clock synchronized: yes
              NTP service: n/a
          RTC in local TZ: no
```

10. Start `cam-app-example` in calibration mode from the DomU2 terminal:

```
cam-app-example -u 22085ddc-bc10-11ed-9a44-7ef9696e -t 2000 -c 5 -s 2 -C
```

The stream event log files (`.csel`) for each stream are generated. The output should look as below:

```
Cam application configuration:
    Service IP address: 127.0.0.1
    Service port: 21604
    UUID base: 22085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 2
    Processing period (ms): 2000
    Processing count: 5
    Multiple connection support: false
    Calibration mode support: true
    Calibration directory: ./[uuid].csel
    Fault injection support: false
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
    Stream 0 sends event 0
    Stream 1 sends event 0
    ...
```

List the files generated:


```
ls -1 *.csel
```

The stream event log files can be shown as below:

```
22085ddc-bc10-11ed-9a44-7ef9696e0000.csel
22085ddc-bc10-11ed-9a44-7ef9696e0001.csel
```

11. Run `cam-tool` from the DomU2 terminal to analyze stream event log files and convert them to stream configuration files (`.csc.yml`).

```
cam-tool analyze -m 1000000 -i 22085ddc-bc10-11ed-9a44-7ef9696e0000.csel
```

The analysis result is reported from the DomU2 terminal as below, the timeout value might change:

```
CAM event log analyze report:
Input event log file:                   22085ddc-bc10-11ed-9a44-7ef9696e0000.csel
Output configuration file:              analyzed.csc.yml
Stream UUID:                            22085ddc-bc10-11ed-9a44-7ef9696e0000
Stream name:                            CAM STREAM  0
Timeout between init and start:         300000
Timeout between start and event:        450000
Application running times:              1
Processing count in each run:           [5]

Event ID        timeout
0               3000001
```

The stream configuration files contain human-readable settings used for the deployment phase of a critical application. Users can modify this configuration, for example to fine tune timeout values depending on the system capabilities.

Run `cam-tool` for the other stream.

```
cam-tool analyze -m 1000000 -i 22085ddc-bc10-11ed-9a44-7ef9696e0001.csel
```

Then, use the `cam-tool pack` command for each of the streams to generate deployment data.

```
cam-tool pack -i 22085ddc-bc10-11ed-9a44-7ef9696e0000.csc.yml
cam-tool pack -i 22085ddc-bc10-11ed-9a44-7ef9696e0001.csc.yml
```

12. Run the `cam-tool deploy` command from the DomU2 terminal to transfer the generated stream deployment data to SI Cluster 1 (where `cam-service` is running):

```
cam-tool deploy -i 22085ddc-bc10-11ed-9a44-7ef9696e0000.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

After that, the stream data of `22085ddc-bc10-11ed-9a44-7ef9696e0000` is deployed to the `cam-service` file system.

Running `cam-tool deploy` one more time deploys the data of the other stream to `cam-service`.

```
cam-tool deploy -i 22085ddc-bc10-11ed-9a44-7ef9696e0001.csd -a 192.168.1.1
```

The output on the SI Cluster 1 terminal after each one of the `cam-tool deploy` command should look as below, the connection number might change:

```
Connection 4 is created.
Deploy Message

Connection 4 is closed.
```

13. List all the files from the SI Cluster 1 terminal:

```
fs ls RAM:/
```

The stream deployment data can be shown as below:

```
11085ddc-bc10-11ed-9a44-7ef9696e0000.csd
11085ddc-bc10-11ed-9a44-7ef9696e0001.csd
11085ddc-bc10-11ed-9a44-7ef9696e0002.csd
11085ddc-bc10-11ed-9a44-7ef9696e0003.csd
22085ddc-bc10-11ed-9a44-7ef9696e0000.csd
22085ddc-bc10-11ed-9a44-7ef9696e0001.csd
```

14. Start `cam-app-example` from the DomU1 terminal to create an application with four streams. Each stream sends an event message 10 times with a period of 3000 milliseconds.

```
cam-app-example -u 11085ddc-bc10-11ed-9a44-7ef9696e -t 3000 -c 10 -s 4 -a 192.168.1.
↪1
```

The following configure messages are expected from the Primary Compute terminal:

```
Cam application configuration:
    Service IP address: 192.168.1.1
    Service port: 21604
    UUID base: 11085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 4
    Processing period (ms): 3000
    Processing count: 10
    Multiple connection support: false
    Calibration mode support: false
    Fault injection support: false
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
Starting activity...
Starting activity...
```

And the log of sent event messages are shown repeatedly:

```
Stream 0 sends event 0
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
```

(continues on next page)

(continued from previous page)

```
Stream 0 sends event 0
Stream 1 sends event 0
Stream 2 sends event 0
Stream 3 sends event 0
...
```

While `cam-app-example` is running on DomU1, start another instance on DomU2. `cam-app-example` has a mode to inject a fault to test the CAM framework. Run `cam-app-example` again from the DomU2 terminal with fault injection to event stream 0:

```
cam-app-example -u 22085ddc-bc10-11ed-9a44-7ef9696e -t 2000 -c 5 -s 2 -f -S 0 -T␣
↪1000 -a 192.168.1.1
```

The following configure messages are expected from the Primary Compute terminal:

```
Cam application configuration:
    Service IP address: 192.168.1.1
    Service port: 21604
    UUID base: 22085ddc-bc10-11ed-9a44-7ef9696e
    Stream count: 2
    Processing period (ms): 2000
    Processing count: 5
    Multiple connection support: false
    Calibration mode support: false
    Fault injection support: true
    Fault injection time: 1000
    Fault injection stream: 0
    Event(s) interval time (ms): 0
Using libcam v1.0
Starting activity...
Starting activity...
```

And the log of sent event messages are shown repeatedly:

```
Stream 0 sends event 0
Stream 1 sends event 0
Stream 1 sends event 0
Stream 1 sends event 0
...

As observed from the SI Cluster 1 terminal, ``cam-service`` is loading
four stream deployment files from DomU1 and two stream deployment files from
DomU2 for monitoring. In the following log, the stream messages are received
and processed by it:

.. code-block:: text

   Connection 4 is created.
   Init Message
   Stream 11085ddc-bc10-11ed-9a44-7ef9696e0002 configuration is loaded.
   Init Message
   Stream 11085ddc-bc10-11ed-9a44-7ef9696e0001 configuration is loaded.
   Init Message
```

(continues on next page)

(continued from previous page)

```
   Stream 11085ddc-bc10-11ed-9a44-7ef9696e0003 configuration is loaded.
   Init Message
   Stream 11085ddc-bc10-11ed-9a44-7ef9696e0000 configuration is loaded.
   Start Message
   Start Message
   Start Message
   Start Message
   Event Message
   Event Message
   Event Message
   Event Message

   Connection 5 is created.
   Init Message
   Stream 22085ddc-bc10-11ed-9a44-7ef9696e0001 configuration is loaded.
   Init Message
   Stream 22085ddc-bc10-11ed-9a44-7ef9696e0000 configuration is loaded.
   Start Message
   Start Message
   Event Message
   Event Message
   Event Message
   # Repeated event messages
   ...

The fault happens 1000ms after stream initialization. At that time
``cam-service`` should detect a stream temporal error with the following
output from the SI Cluster 1 terminal.

.. code-block:: text

   # Repeated event messages
   ...
   ERROR: Stream temporal error:
   ERROR:     stream_name: CAM STREAM  0
   ERROR:     stream_uuid: 2285ddc-bc10-11ed-9a44-7ef9696e0000
   ERROR:     event_id: 0
   ERROR:     time_received: 0
   ERROR:     time_expected: 1710275907816057
   # Repeated event messages
   ...
   ERROR: Stream state error:
   ERROR:     stream_name: CAM STREAM  0
   ERROR:     stream_uuid: 22085ddc-bc10-11ed-9a44-7ef9696e0000
   ERROR:     timestamp: 1710275909816069
   ERROR:     current_state: Failed state
   ERROR:     requested_state: In-progress state
```

Note: `time_received:  0` should be ignored as the `time_received` is not set during a fault.

15. To leave the DomU1 console, type `Ctrl-]` and enter `send esc`.

16. To leave the DomU2 console, type `Ctrl-]`.
17. To shut down the FVP and terminate the emulation automatically, follow the below steps:
    - Issue the following command on the Primary Compute terminal.

      ```
      shutdown now
      ```

      The below messages indicate the shutdown process is complete.

      ```
      [  OK  ] Finished System Power Off.
      [  OK  ] Reached target System Power Off.
      reboot: Power down
      ```

    - Close the tmux pane started for DomU2 by pressing `Ctrl-d`.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Critical Application Monitoring Demo` as `Use-Case`.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 65 minutes to complete. See *Integration Tests Validating the Critical Application Monitoring Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_40_cam.CAMServiceTest.test_cam_service_boot_on_si: PASSED (0.00s)
RESULTS - test_40_cam.CAMTest.test_cam_app_example_to_service_on_si: PASSED (63.55s)
RESULTS - test_40_cam.CAMTest.test_cam_app_example_to_service_on_si_with_multiple_
↪connections: PASSED (62.12s)
RESULTS - test_40_cam.CAMTest.test_cam_ptp_sync: PASSED (143.39s)
RESULTS - test_40_cam.CAMTest.test_cam_tool_deploy_to_si: PASSED (63.58s)
RESULTS - test_40_cam.CAMTest.test_cam_tool_pack: PASSED (62.96s)
RESULTS - test_40_cam.CAMTest.test_data_calibration: PASSED (175.01s)
RESULTS - test_40_cam.CAMTest.test_logical_check_on_si: PASSED (26.43s)
RESULTS - test_40_cam.CAMTest.test_temporal_check_on_si: PASSED (78.28s)
RESULTS - test_40_cam.CAMTestDomU2.test_cam_app_example_to_service_on_si: PASSED (60.84s)
RESULTS - test_40_cam.CAMTestDomU2.test_cam_app_example_to_service_on_si_with_multiple_
↪connections: PASSED (63.07s)
RESULTS - test_40_cam.CAMTestDomU2.test_cam_ptp_sync: PASSED (13.27s)
RESULTS - test_40_cam.CAMTestDomU2.test_cam_tool_deploy_to_si: PASSED (32.96s)
RESULTS - test_40_cam.CAMTestDomU2.test_cam_tool_pack: PASSED (32.19s)
RESULTS - test_40_cam.CAMTestDomU2.test_data_calibration: PASSED (100.73s)
RESULTS - test_40_cam.CAMTestDomU2.test_logical_check_on_si: PASSED (25.67s)
RESULTS - test_40_cam.CAMTestDomU2.test_temporal_check_on_si: PASSED (77.41s)
```

(continues on next page)

(continued from previous page)

```
RESULTS - test_40_cam.CAMTestMultiDom.test_cam_app_example_to_service_on_si_with_
↪multiple_vms: PASSED (68.86s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Safety Island Actuation Demo

The demo can be run on the Baremetal Architecture or Virtualization Architecture. See *Safety Island Actuation Demo* for further details.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

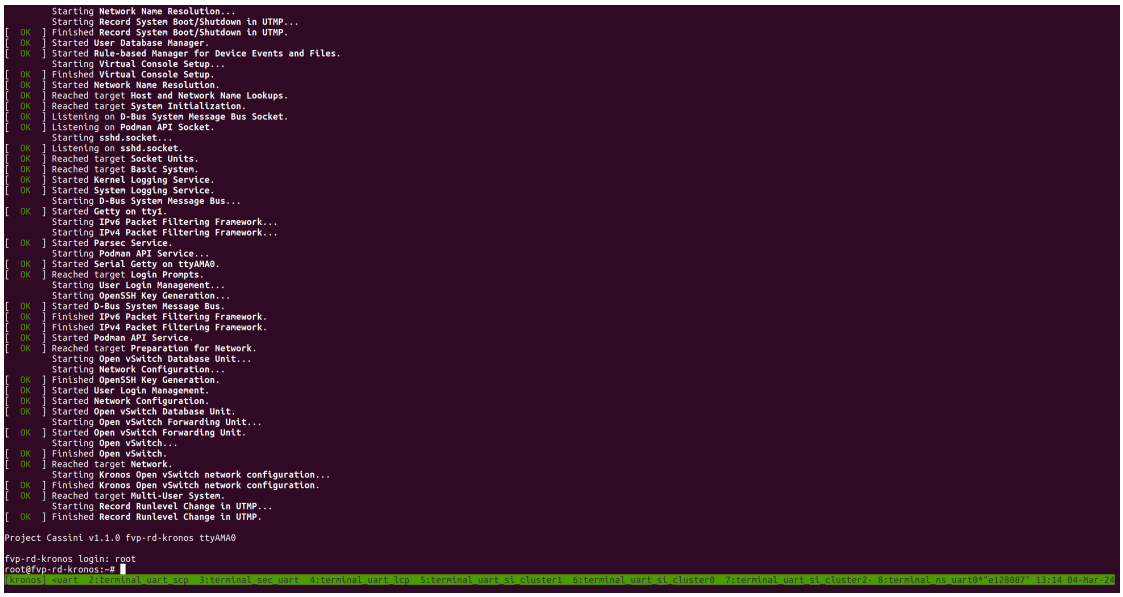
1. Select `Safety Island Actuation Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The user should wait for the system to boot and for the Linux prompt to appear. The following image shows an example on how the terminal should look after the fvp invocation.

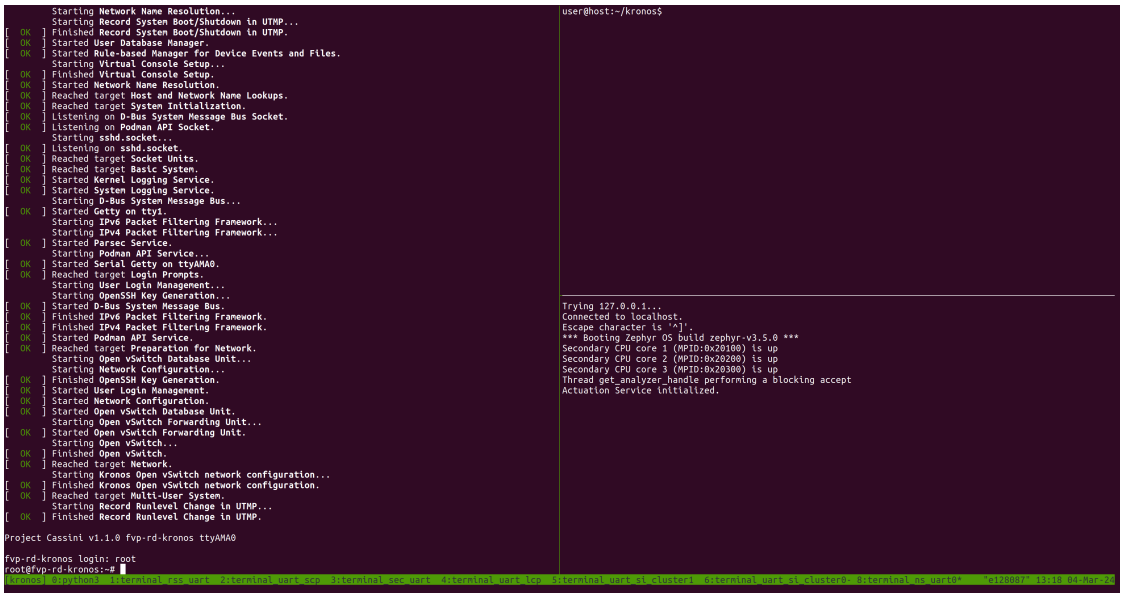


The Safety Island (SI) Cluster 2 terminal running the Actuation Service is available via the tmux window titled `terminal_uart_si_cluster2`. For ease of navigation, it's recommended to join the SI Cluster 2 terminal window to the Primary Compute terminal window and creating a tmux pane attached to the build host machine in order to issue commands on it.

Follow the steps below to achieve the same:

1. Ensure that the tmux window titled `terminal_ns_uart0` is selected. If not, press `Ctrl-b w` from the tmux session, navigate to the tmux window titled `terminal_ns_uart0` using the arrow keys, then press the `Enter` key.
2. Press `Ctrl-b %` to add a new tmux pane which will be used to issue commands on the build host machine.
3. Press `Ctrl-b :` and then type `join-pane -s :terminal_uart_si_cluster2` followed by pressing `Enter` key to join the Actuation Service terminal window to the Primary Compute terminal window.

Refer to the following image of the tmux panes rearrangement. Panes can be navigated using `Ctrl-b` followed by the arrow keys.



The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Run the Demo

1. Run the `ping` command from the Primary Compute terminal (running Linux) to verify that it can communicate with the Safety Island Cluster 2 (running Zephyr):

   ```
   ping 192.168.2.1 -c 10
   ```

   The output should look like the following line, repeated 10 times:

   ```
   64 bytes from 192.168.2.1 seq=0 ttl=64 time=0.151 ms
   ```

2. From the tmux pane started for the build host machine terminal, start the Packet Analyzer:

   ```
   cd ~/kronos/
   kas shell -c "oe-run-native packet-analyzer-native start_analyzer -L debug -a␣
   ↪localhost -c ./data"
   ```

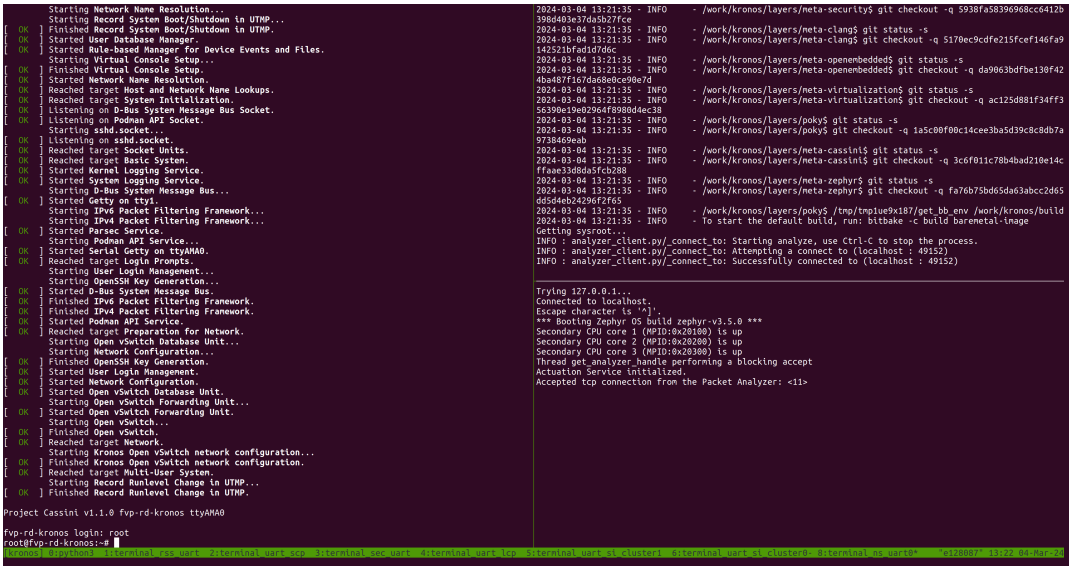
   The following messages are expected from the host terminal:

   ```
   INFO : analyzer_client.py/_connect_to: Starting analyze, use Ctrl-C to stop the␣
   ↪process.
   INFO : analyzer_client.py/_connect_to: Attempting a connect to (localhost : 49152)
   INFO : analyzer_client.py/_connect_to: Successfully connected to (localhost : 49152)
   ```


A message similar to the following should appear on the SI Cluster 2 terminal:

```
Actuation Service initialized.
Accepted tcp connection from the Packet Analyzer: <11>
```

Refer to the following image for an invocation example of the Packet Analyzer.



3. Start the Player on the Primary Compute terminal which replays a recording of a driving scenario:

```
actuation_player -p /usr/share/actuation_player/
```

A message similar to the following should appear on the Primary Compute terminal:

A message similar to the following should appear on the SI Cluster 2 terminal:

```
51572682601: -0.0000 (m/s^2) |  0.0000 (rad)
51597466928: -0.0000 (m/s^2) |  0.0000 (rad)
51622532911: -0.0000 (m/s^2) |  0.0000 (rad)
51647642316: -0.0000 (m/s^2) |  0.0000 (rad)
51672535849: -0.0000 (m/s^2) |  0.0000 (rad)
51697376579: -0.0000 (m/s^2) |  0.0000 (rad)
51722500414: -0.0000 (m/s^2) |  0.0000 (rad)
51747622543: -0.0000 (m/s^2) |  0.0000 (rad)
51772496466: -0.0000 (m/s^2) |  0.0000 (rad)
Thread get_analyzer_handle performing a blocking accept
```

A message similar to the following should appear on the host terminal where the Packet Analyzer is running:

```
INFO : analyzer_client.py/_connect_to: Starting analyze, use Ctrl-C to stop the␣
↪process.
INFO : analyzer_client.py/_connect_to: Attempting a connect to (localhost : 49152)
INFO : analyzer_client.py/_connect_to: Successfully connected to (localhost : 49152)
```

(continues on next page)

(continued from previous page)

```
INFO : analyzer_client.py/run_analyze_on_chain: (1) Analyzer synced with packet␣
↪chain
```

The following messages should appear, but values may differ once the Packet Analyzer has finished running:

```
INFO : analyzer_client.py/run_analyze_on_chain: All expected control packets␣
↪received
INFO : analyzer_client.py/_log_jitter: Observed Frequency = 21.36147200, Avg Jitter␣
↪= 0.02624593, Std Deviation:0.06096328
INFO : analyzer_client.py/run_analyze_on_chain: End of cycle: AnalyzerResult.SUCCESS

INFO : analyzer_client.py/_tear_conn: Received fin ack from Actuation Service

Chain ID   Result
0          AnalyzerResult.SUCCESS
```

4. To shut down the FVP and terminate the emulation automatically, follow the below steps:

   - Issue the following command on the Primary Compute terminal.

     ```
     shutdown now
     ```

     The below messages indicate the shutdown process is complete.

     ```
     [  OK  ] Finished System Power Off.
     [  OK  ] Reached target System Power Off.
     reboot: Power down
     ```

   - Close the tmux pane started for the build host machine by pressing `Ctrl-d`.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Actuation Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 45 minutes to complete. See *Integration Tests Validating the Safety Island Actuation Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_30_actuation.ActuationTest.test_analyzer_help: PASSED (0.44s)
RESULTS - test_30_actuation.ActuationTest.test_ping: PASSED (26.04s)
RESULTS - test_30_actuation.ActuationTest.test_player_to_analyzer: PASSED (167.15s)
```


Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Virtualization Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Virtualization Architecture image:

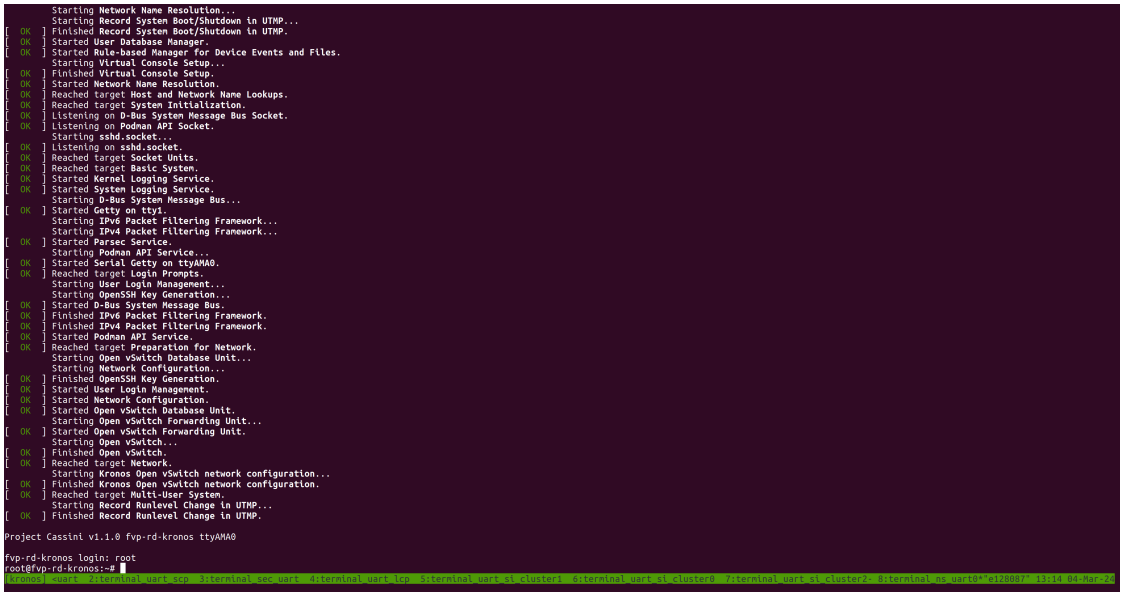
1. Select `Safety Island Actuation Demo` from the `Use-Case` menu.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The user should wait for the system to boot and for the Linux prompt to appear. On a Virtualization Architecture image, this will access the Dom0 terminal. The following image shows an example on how the terminal should look after the fvp invocation.

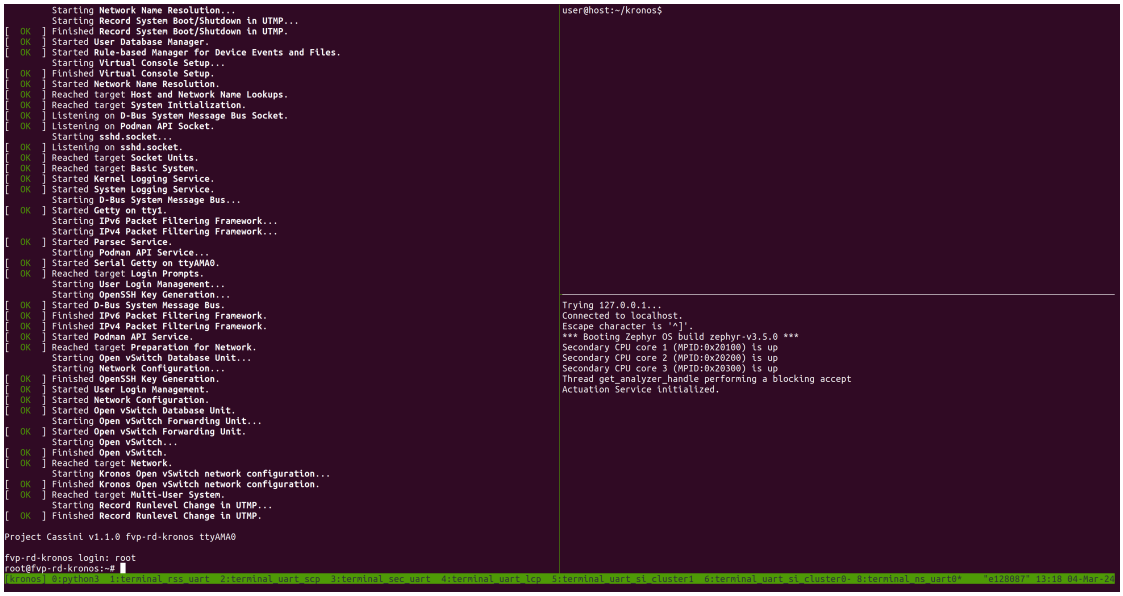


The Safety Island (SI) Cluster 2 terminal running the Actuation Service is available via the tmux window titled `terminal_uart_si_cluster2`. For ease of navigation, it's recommended to join the SI Cluster 2 terminal window to the Primary Compute terminal window and creating a tmux pane attached to the build host machine in order to issue commands on it.

Follow the steps below to achieve the same:

1. Ensure that the tmux window titled `terminal_ns_uart0` is selected. If not, press `Ctrl-b w` from the tmux session, navigate to the tmux window titled `terminal_ns_uart0` using the arrow keys, then press the `Enter` key.
2. Press `Ctrl-b %` to add a new tmux pane which will be used to issue commands on the build host machine.
3. Press `Ctrl-b :` and then type `join-pane -s :terminal_uart_si_cluster2` followed by pressing `Enter` key to join the Actuation Service terminal window to the Primary Compute terminal window.

Refer to the following image of the tmux panes rearrangement. Panes can be navigated using `Ctrl-b` followed by the arrow keys.



The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Note: A message similar to the following might appear in the Primary Compute terminal `(XEN) d2v0: vGICR: SGI: unhandled word write 0x000000ffffffff to ICACTIVER0`, this is an expected behavior.

Run the Demo

1. Enter the DomU1 console using the `xl` tool:

   ```
   xl console domu1
   ```

   DomU1 can be logged into as `root` user without a password in the Linux terminal. This command will provide a console on the DomU1. To exit, enter `Ctrl-]` (to access the FVP telnet shell), followed by typing `send esc` into the telnet shell and pressing `Enter`. See the xl documentation for further details.

2. Run the `ping` command from the DomU1 terminal (running Linux) to verify that it can communicate with the Safety Island Cluster 2 (running Zephyr):

   ```
   ping 192.168.2.1 -c 10
   ```

   The output should look like the following line, repeated 10 times:

   ```
   64 bytes from 192.168.2.1 seq=0 ttl=64 time=0.151 ms
   ```

3. From the tmux pane started for the build host machine terminal, start the Packet Analyzer:

   ```
   cd ~/kronos/
   kas shell -c "oe-run-native packet-analyzer-native start_analyzer -L debug -a␣
   ↪localhost -c ./data"
   ```

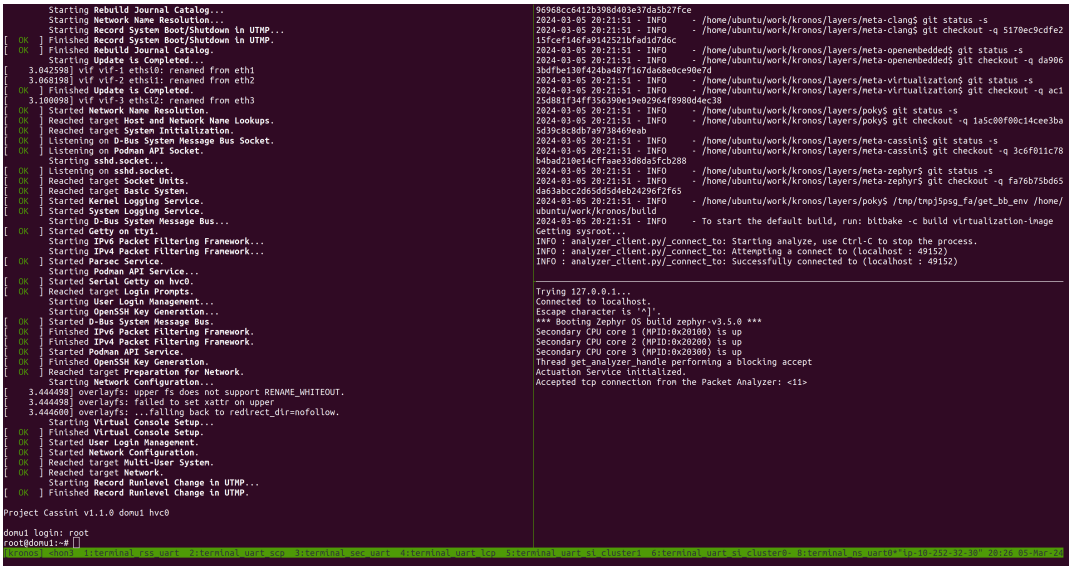
   The following messages are expected from the host terminal:

   ```
   INFO : analyzer_client.py/_connect_to: Starting analyze, use Ctrl-C to stop the␣
   ↪process.
   INFO : analyzer_client.py/_connect_to: Attempting a connect to (localhost : 49152)
   INFO : analyzer_client.py/_connect_to: Successfully connected to (localhost : 49152)
   ```

   A message similar to the following should appear on the SI Cluster 2 terminal:

   ```
   Actuation Service initialized.
   Accepted tcp connection from the Packet Analyzer: <11>
   ```

   Refer to the following image for an invocation example of the Packet Analyzer.



4. Start the Player on DomU1 which replays a recording of a driving scenario:

```
actuation_player -p /usr/share/actuation_player/
```

A message similar to the following should appear on the Primary Compute terminal:

A message similar to the following should appear on the SI Cluster 2 terminal:

```
51572682601: -0.0000 (m/s^2) |  0.0000 (rad)
51597466928: -0.0000 (m/s^2) |  0.0000 (rad)
51622532911: -0.0000 (m/s^2) |  0.0000 (rad)
51647642316: -0.0000 (m/s^2) |  0.0000 (rad)
51672535849: -0.0000 (m/s^2) |  0.0000 (rad)
51697376579: -0.0000 (m/s^2) |  0.0000 (rad)
51722500414: -0.0000 (m/s^2) |  0.0000 (rad)
51747622543: -0.0000 (m/s^2) |  0.0000 (rad)
51772496466: -0.0000 (m/s^2) |  0.0000 (rad)
Thread get_analyzer_handle performing a blocking accept
```

A message similar to the following should appear on the host terminal where the Packet Analyzer is running:

```
INFO : analyzer_client.py/_connect_to: Starting analyze, use Ctrl-C to stop the␣
↪process.
INFO : analyzer_client.py/_connect_to: Attempting a connect to (localhost : 49152)
INFO : analyzer_client.py/_connect_to: Successfully connected to (localhost : 49152)
INFO : analyzer_client.py/run_analyze_on_chain: (1) Analyzer synced with packet␣
↪chain
```

The following messages should appear, but values may differ once the Packet Analyzer has finished running:

```
INFO : analyzer_client.py/run_analyze_on_chain: All expected control packets␣
↪received
```

(continues on next page)

(continued from previous page)

```
INFO : analyzer_client.py/_log_jitter: Observed Frequency = 21.36147200, Avg Jitter␣
↪= 0.02624593, Std Deviation:0.06096328
INFO : analyzer_client.py/run_analyze_on_chain: End of cycle: AnalyzerResult.SUCCESS

INFO : analyzer_client.py/_tear_conn: Received fin ack from Actuation Service

Chain ID   Result
0          AnalyzerResult.SUCCESS
```

5. To leave the DomU1 console, type `Ctrl-]` and enter `send esc`.
6. To shut down the FVP and terminate the emulation automatically, follow the below steps:
   - Issue the following command on the Primary Compute terminal.

     ```
     shutdown now
     ```

     The below messages indicate the shutdown process is complete.

     ```
     [  OK  ] Finished System Power Off.
     [  OK  ] Reached target System Power Off.
     reboot: Power down
     ```

   - Close the tmux pane started for the build host machine by pressing `Ctrl-d`.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Actuation Demo` as `Use-Case`.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 110 minutes to complete. See *Integration Tests Validating the Safety Island Actuation Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_30_actuation.ActuationTest.test_analyzer_help: PASSED (0.40s)
RESULTS - test_30_actuation.ActuationTest.test_ping: PASSED (42.15s)
RESULTS - test_30_actuation.ActuationTest.test_player_to_analyzer: PASSED (265.77s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Safety Island Communication Demo (using HIPC)

The Safety Island Communication Demo uses *HIPC (Heterogeneous Inter-processor Communication)* to validate networking between the Primary Compute and the three Safety Island clusters.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Safety Island Communication Demo (using HIPC)` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Communication Demo (using HIPC)` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 45 minutes to complete. See *Integration Tests Validating the Safety Island Communication Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster0: PASSED (223.04s)
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster1: PASSED (351.62s)
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster2: PASSED (426.17s)
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster_cl0_cl1: PASSED (100.46s)
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster_cl0_cl2: PASSED (120.00s)
RESULTS - test_30_hipc.HIPCTestBase.test_hipc_cluster_cl1_cl2: PASSED (161.80s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cl0_cl1: PASSED (37.34s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cl0_cl2: PASSED (36.80s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cl1_cl2: PASSED (37.10s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cluster0: PASSED (102.31s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cluster1: PASSED (95.50s)
RESULTS - test_30_hipc.HIPCTestBase.test_ping_cluster2: PASSED (92.91s)
RESULTS - test_30_ptp.PTPTest.test_ptp_linux_services: PASSED (4.56s)
RESULTS - test_30_ptp.PTPTest.test_ptp_si_clients: PASSED (34.73s)
```


Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Virtualization Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Virtualization Architecture image:

1. Select `Safety Island Communication Demo (using HIPC)` as `Use-Case`.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Communication Demo (using HIPC)` as `Use-Case`.
2. Select `Virtualization` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 90 minutes to complete. See *Integration Tests Validating the Safety Island Communication Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster0: PASSED (311.77s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster1: PASSED (389.55s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster2: PASSED (413.83s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster_cl0_cl1: PASSED␣
↪(114.97s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster_cl0_cl2: PASSED␣
↪(138.32s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_hipc_cluster_cl1_cl2: PASSED␣
↪(182.33s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cl0_cl1: PASSED (60.64s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cl0_cl2: PASSED (61.63s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cl1_cl2: PASSED (61.06s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cluster0: PASSED (153.76s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cluster1: PASSED (154.39s)
```

(continues on next page)

(continued from previous page)

```
RESULTS - test_30_hipc_virtualization.HIPCTestDomU1.test_ping_cluster2: PASSED (153.34s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU2.test_hipc_cluster1: PASSED (365.17s)
RESULTS - test_30_hipc_virtualization.HIPCTestDomU2.test_ping_cluster1: PASSED (152.43s)
RESULTS - test_30_ptp.PTPTest.test_ptp_linux_services: PASSED (7.17s)
RESULTS - test_30_ptp.PTPTest.test_ptp_si_clients: PASSED (47.37s)
RESULTS - test_30_ptp.PTPTestDomU1.test_ptp_domu_client: PASSED (50.57s)
RESULTS - test_30_ptp.PTPTestDomU1.test_ptp_linux_services: PASSED (1.38s)
RESULTS - test_30_ptp.PTPTestDomU2.test_ptp_domu_client: PASSED (50.31s)
RESULTS - test_30_ptp.PTPTestDomU2.test_ptp_linux_services: PASSED (1.48s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Parsec-enabled TLS Demo

The demo can be run on the Baremetal Architecture. It consists of a TLS server and a TLS client. Refer to *Parsec-enabled TLS Demo* for more information on this application. This demo is included as part of the `Safety Island Actuation Demo`.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Safety Island Actuation Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The user should wait for the system to boot and for the Linux prompt to appear.

The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Run the Demo

The demo consists of a TLS server and a TLS client. Refer to *Parsec-enabled TLS Demo* for more information on this application.

1. Run `ssl_server` from the Primary Compute terminal in the background and press the `Enter` key to continue:

```
ssl_server &
```

   A message similar to the following should appear:

```
. Seeding the random number generator... ok
. Loading the server cert. and key... ok
. Bind on https://localhost:4433/ ... ok
. Setting up the SSL data.... ok
. Waiting for a remote connection ...
```

   The TLS client can take an optional parameter as the TLS server IP address. The default value of the parameter is `localhost`.

2. Run `ssl_client1` from the Primary Compute terminal in a container:

```
docker run  --rm -v /run/parsec/parsec.sock:/run/parsec/parsec.sock -v /usr/bin/ssl_
↪client1:/usr/bin/ssl_client1 --network host docker.io/library/ubuntu:22.04 ssl_
↪client1
```

   After a few seconds, a message similar to the following should appear:

```
  . Seeding the random number generator... ok
  . Loading the CA root certificate ... ok (0 skipped)
  . Connecting to tcp/localhost/4433... ok
  . Setting up the SSL/TLS structure... ok
  . Performing the SSL/TLS handshake... ok
  . Verifying peer X.509 certificate... ok
  > Write to server: 18 bytes written

GET / HTTP/1.0

< Read from server: 156 bytes read

HTTP/1.0 200 OK
Content-Type: text/html

<h2>mbed TLS Test Server</h2>
<p>Successful connection using: TLS-ECDHE-RSA-WITH-CHACHA20-POLY1305-SHA256</p>
```

3. Stop the TLS server and synchronize the container image to the persistent storage:

```
pkill ssl_server
sync
```

4. To shut down the FVP and terminate the emulation automatically, issue the following command on the Primary Compute terminal.

```
shutdown now
```

The below messages indicate the shutdown process is complete.

```
[  OK  ] Finished System Power Off.
[  OK  ] Reached target System Power Off.
reboot: Power down
```

Automated Validation

For more details about the validation of Parsec demo, refer to *Integration Tests Validating the Parsec-enabled TLS Demo*.

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Actuation Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 45 minutes to complete. See *Integration Tests Validating the Parsec-enabled TLS Demo* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_40_parsec.ParsecTest.test_parsec_demo: PASSED (479.25s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Primary Compute PSA Protected Storage and Crypto APIs Architecture Test Suite

The demo can be run on the Baremetal Architecture. Refer to *Primary Compute Secure Services* for more information on this application. This demo is included as part of the `Critical Application Monitoring Demo`.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Critical Application Monitoring Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The user should wait for the system to boot and for the Linux prompt to appear.

The Reference Software Stack running on the Primary Compute can be logged into as `root` user without a password in the Linux terminal. Run the below command to guarantee that all the expected services have been initialized.

```
systemctl is-system-running --wait
```

Wait for it to return. The expected terminal output is `running`.

Run the Demo

The demo consists of simple tests run from the Linux terminal. Refer to *Primary Compute Secure Services* for more information on this application.

1. Run the PSA Crypto API tests from the Primary Compute terminal using the following command:

   ```
   psa-crypto-api-test
   ```

   A message similar to the following should appear once the tests have completed:

   ```
   ************ Crypto Suite Report **********
   TOTAL TESTS     : 59
   TOTAL PASSED    : 59
   TOTAL SIM ERROR : 0
   TOTAL FAILED    : 0
   TOTAL SKIPPED   : 0
   ****************************************
   ```

2. Run the PSA Protected Storage API tests from the Primary Compute terminal using the following command:

   ```
   psa-ps-api-test
   ```

   A message similar to the following should appear once the tests have completed:

   ```
   ************ Storage Suite Report **********
   TOTAL TESTS     : 17
   TOTAL PASSED    : 11
   TOTAL SIM ERROR : 0
   TOTAL FAILED    : 0
   TOTAL SKIPPED   : 6
   ****************************************
   ```

3. To shut down the FVP and terminate the emulation automatically, issue the following command on the Primary Compute terminal.

   ```
   shutdown now
   ```

   The below messages indicate the shutdown process is complete.

```
[  OK  ] Finished System Power Off.
[  OK  ] Reached target System Power Off.
reboot: Power down
```

Automated Validation

For more details about the validation of PSA Architecture Test Suite, refer to *Integration Tests Validating Primary Compute PSA APIs Architecture Test Suite*.

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Critical Application Monitoring Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 40 minutes to complete. See *Integration Tests Validating Primary Compute PSA APIs Architecture Test Suite* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_50_trusted_services.KronosTrustedServices.test_03_psa_crypto_api_test:␣
↪PASSED (298.70s)
RESULTS - test_50_trusted_services.KronosTrustedServices.test_05_psa_ps_api_test: PASSED␣
↪(68.88s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Safety Island PSA Secure Storage APIs Architecture Test Suite

The demo can be run on the Baremetal Architecture. See *Safety Island PSA Architecture Test Suite* for further details.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Safety Island PSA Secure Storage APIs Architecture Test Suite` from the `Use-Case` menu.

2. Select `Build`.

Run the FVP

To start the FVP:

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The Safety Island Cluster 2 terminal running the `PSA Secure Storage APIs Architecture Test Suite` is available via the tmux window titled `terminal_uart_si_cluster2`.

The Safety Island Cluster 2 tmux window can be accessed by typing `Ctrl-b w`, using the arrow keys to select `terminal_uart_si_cluster2` then pressing the `Enter` key.

Run the Demo

The tests will automatically run. A log similar to the following should be visible; it is normal for some tests to be skipped but there should be no failed tests:

```
***** PSA Architecture Test Suite - Version 1.4 *****
Running.. Storage Suite
****************************************
TEST: 401 | DESCRIPTION: UID not found check | UT: STORAGE
[Info] Executing tests from non-secure
[Info] Executing ITS Tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
Set storage for UID 6
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

[Info] Executing PS Tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
Set storage for UID 6
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

TEST RESULT: PASSED

****************************************
```

(continues on next page)

(continued from previous page)

```
<further tests removed from log for brevity>

************ Storage Suite Report **********
TOTAL TESTS     : 17
TOTAL PASSED    : 11
TOTAL SIM ERROR : 0
TOTAL FAILED    : 0
TOTAL SKIPPED   : 6
****************************************
```

To shut down the FVP and terminate the emulation, select the terminal titled as `python3` where the `runfvp` was launched by pressing `Ctrl-b 0` and press `Ctrl-c` to stop the FVP process.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island PSA Secure Storage APIs Architecture Test Suite` from the `Use-Case` menu.
2. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
3. Select `Build`.

The complete test suite takes around 15 minutes to complete. See *Integration Tests Validating Safety Island PSA APIs Architecture Test Suite* for more details.

The following message is expected in the output to validate this Use-Case:

```
RESULTS - test_10_si_psa_arch_tests.SIPSAArchTests.test_psa_si_cluster2: PASSED (0.00s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Safety Island PSA Crypto APIs Architecture Test Suite

The demo can be run on the Baremetal Architecture. See *Safety Island PSA Architecture Test Suite* for further details.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Safety Island PSA Crypto APIs Architecture Test Suite` from the `Use-Case` menu.
2. Select `Build`.

Run the FVP

To start the FVP:

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The `PSA Crypto APIs Architecture Test Suite` is deployed on all the 3 Safety Island (SI) Clusters. The test results can be seen on the following tmux windows:

- `terminal_uart_si_cluster0`
- `terminal_uart_si_cluster1`
- `terminal_uart_si_cluster2`

The user can navigate through the windows mentioned above by pressing `Ctrl-b w` and arrow keys followed by the `Enter` key.

Run the tests

The tests will automatically run after the FVP is started. The complete test suite takes around 5 minutes to complete. When the tests finish, a log similar to the following should be visible. Normally no failure should be seen:

```
*********** Crypto Suite Report **********
TOTAL TESTS     : 61
TOTAL PASSED    : 61
TOTAL SIM ERROR : 0
TOTAL FAILED    : 0
TOTAL SKIPPED   : 0
*****************************************
```

To shut down the FVP and terminate the emulation, select the terminal titled as `python3` where the `runfvp` was launched by pressing `Ctrl-b 0` and press `Ctrl-c` to stop the FVP process.

Note: This use-case does not require waiting for the Primary Compute to boot.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island PSA Crypto APIs Architecture Test Suite` from the `Use-Case` menu.
2. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
3. Select `Build`.

The complete test suite takes around 35 minutes to complete. See *Integration Tests Validating Safety Island PSA APIs Architecture Test Suite* for more details.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_10_si_psa_arch_tests.SIPSAArchTests.test_psa_si_cluster0: PASSED (0.01s)
RESULTS - test_10_si_psa_arch_tests.SIPSAArchTests.test_psa_si_cluster1: PASSED (0.01s)
RESULTS - test_10_si_psa_arch_tests.SIPSAArchTests.test_psa_si_cluster2: PASSED (0.01s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Fault Management Demo

The demo uses the Safety Island Cluster 1 console and it can be run on the Baremetal Architecture of the Safety Island Actuation Demo. Refer to *Fault Management* for further details.

Baremetal Architecture

Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build the Baremetal Architecture image:

1. Select `Safety Island Actuation Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP:

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The Fault Management subsystem is deployed on Safety Island Cluster 1 so the instructions below should be executed on its terminal.

The Safety Island Cluster 1 tmux window can be accessed by typing `Ctrl-b w`, using the arrow keys to select `terminal_uart_si_cluster1` then pressing the `Enter` key.

Run the Demo

The instructions below demonstrate injecting faults into both the System FMU and GIC-720AE FMU and how this affects the SSU safety state.

1. Start by enumerating the configured fault device tree:

   ```
   fault tree
   ```

   The output shows the root fault device `fmu@2a510000` (the System FMU), after which are the attached safety state device `ssu@2a500000` and fault device `fmu@2a570000` (the GIC-720AE FMU):

   ```
   Root 0: fmu@2a510000
           Safety: ssu@2a500000
           Slot 0: fmu@2a570000
   ```

2. After booting, query the initial state of the SSU:

   ```
   fault safety_status ssu@2a500000
   ```

   The initial state is TEST:

   ```
   Status: TEST (0x0)
   ```

3. It is expected that a Fault Management deployment would perform a self-test after boot then signal its outcome to the SSU. For demonstration purposes, simulate a successful self-test completion by issuing the `compl_ok` signal to the SSU:

   ```
   fault safety_control ssu@2a500000 compl_ok
   ```

   The system is now `SAFE` for operation:

   ```
   Signal: compl_ok (0x0)
   State: SAFE (0x3)
   ```

4. Simulate an internal *Lockstep error* (`0x4`) in the System FMU:

   ```
   fault inject fmu@2a510000 0x4
   ```

   Three events are logged:

   - The subsystem reports that it received the fault and that it was non-critical (all System FMU internal faults are non-critical).

- The safety component reports that this caused the SSU to enter the ERRN state.
- The storage component reports that the total historical fault count for this fault on this device is now 1.

```
Injecting fault 0x4 to device fmu@2a510000
[00:04:49.110,000] <inf> fault_mgmt: Fault received (non-critical): 0x4 on␣
↪fmu@2a510000

[00:04:49.110,000] <inf> fault_mgmt_safety: Safety status: ERRN (0x5) on␣
↪ssu@2a500000

[00:04:49.160,000] <inf> fault_mgmt_protected_storage: Fault count for 0x4 on␣
↪fmu@2a510000: 1
```

5. The SSU will remain in the ERRN state until signaled (unless a critical fault occurs). Send a compl_ok signal again to recover from this fault:

```
fault safety_control ssu@2a500000 compl_ok
```

   The SSU is now in the SAFE state again:

```
Signal: compl_ok (0x0)
State: SAFE (0x3)
```

6. Next, inject an *SPI collator external error* (0x20000a00) into the GIC-720AE FMU:

```
fault inject fmu@2a570000 0x20000a00
```

   This results in a similar output to above, except that the received fault was critical and the safety status is now ERRC. (GIC-720AE FMU faults are critical by default, but this can be changed from the shell using the fault set_critical sub-command).

```
Injecting fault 0x20000a00 to device fmu@2a570000
[00:09:13.210,000] <inf> fault_mgmt: Fault received (critical): 0x20000a00 on␣
↪fmu@2a570000

[00:09:13.210,000] <inf> fault_mgmt_safety: Safety status: ERRC (0x6) on␣
↪ssu@2a500000

[00:09:13.270,000] <inf> fault_mgmt_protected_storage: Fault count for 0x20000a00␣
↪on fmu@2a570000: 1
```

7. The number of occurrences of each fault is tracked per device by the storage component. Inject another *Lockstep error* into the System FMU:

```
fault inject fmu@2a510000 0x4
```

   The fault count is now 2. Note that the safety status is still ERRC.

```
Injecting fault 0x4 to device fmu@2a510000
[00:14:02.800,000] <inf> fault_mgmt: Fault received (non-critical): 0x4 on␣
↪fmu@2a510000

[00:14:02.800,000] <inf> fault_mgmt_safety: Safety status:  ERRC (0x6) on␣
↪ssu@2a500000
```

(continues on next page)

(continued from previous page)

```
[00:14:02.860,000] <inf> fault_mgmt_protected_storage: Fault count for 0x4 on␣
↪fmu@2a510000: 2
```

The full list of stored faults can also be queried:

```
fault list
```

This shows all the faults injected into both FMUs above:

```
Fault history:
Fault received (non-critical): 0x4 on fmu@2a510000 : count 2

Fault received (critical): 0x20000a00 on fmu@2a570000 : count 1
```

8. The ERRC represents a critical system failure and cannot be recovered by the software - confirm this by trying to issue `compl_ok` again:

```
fault safety_control ssu@2a500000 compl_ok
```

The SSU status is still ERRC:

```
Signal: compl_ok (0x0)
State: ERRC (0x6)
```

The state can now only be affected through a full system reset (e.g. by stopping and starting the FVP), after which the state will be TEST once again.

9. To shut down the FVP and terminate the emulation, select the terminal titled as `python3` where the `runfvp` was launched by pressing `Ctrl-b 0` and press `Ctrl-c` to stop the FVP process.

See the *Shell Reference* for more details about these and other Fault Management shell sub-commands.

Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Safety Island Actuation Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The complete test suite takes around 45 minutes to complete. See *Integration Tests Validating the Fault Management Subsystem* for more details.

The following messages are expected in the output to validate this Use-Case:

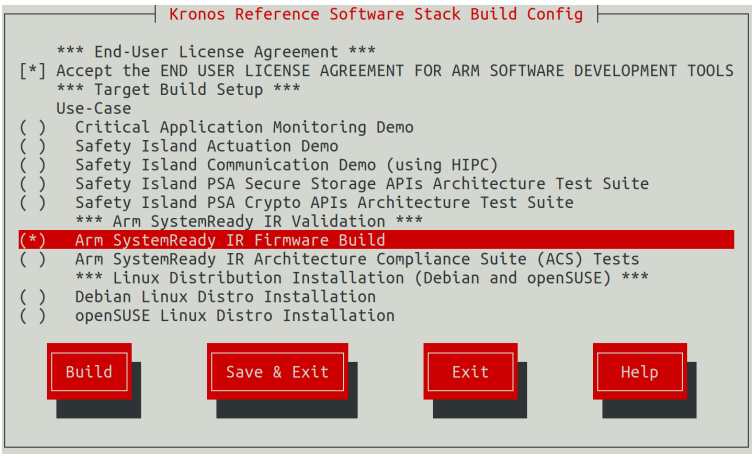

```
RESULTS - test_10_fault_mgmt.FaultMgmtSSUTest.test_ssu_ce_not_ok: PASSED (38.18s)
RESULTS - test_10_fault_mgmt.FaultMgmtSSUTest.test_ssu_compl_ok: PASSED (30.66s)
RESULTS - test_10_fault_mgmt.FaultMgmtSSUTest.test_ssu_nce_ok: PASSED (29.24s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_fmu_fault_clear: PASSED (11.24s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_fmu_fault_count: PASSED (6.04s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_fmu_fault_list: PASSED (36.22s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_fmu_fault_summary: PASSED (22.64s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_gic_fmu_inject: PASSED (18.69s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_system_fmu_internal_inject: PASSED (8.
↪77s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_system_fmu_internal_set_enabled: PASSED␣
↪(11.09s)
RESULTS - test_10_fault_mgmt.FaultMgmtTest.test_tree: PASSED (0.71s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Arm SystemReady IR Validation

Arm SystemReady IR Firmware Build

The Arm SystemReady IR Firmware Build option just builds the Arm SystemReady IR-aligned firmware. Refer to *Arm SystemReady IR* for more details.



Build

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build the Arm SystemReady IR-aligned firmware image:

1. Select `Arm SystemReady IR Firmware Build` under `Arm SystemReady IR Validation` from the `Use-Case` menu.
2. Select `Build`.

The firmware images listed below can be found in the directory `build/tmp_systemready-glibc/deploy/images/fvp-rd-kronos/`.

- `ap-flash-image-fvp-rd-kronos.wic`
- `encrypted_cm_provisioning_bundle_0.bin`
- `encrypted_dm_provisioning_bundle.bin`
- `rss-flash-image-fvp-rd-kronos.wic`
- `rss-nvm-image.bin`
- `rss-rom-image-fvp-rd-kronos.wic.nopt`

Arm SystemReady IR Architecture Compliance Suite (ACS) Tests

The ACS for the Arm SystemReady IR certification is delivered through a live OS image, which enables the basic automation to run the tests.

The system will boot with the ACS live OS image and the ACS tests will run automatically after the system boots. See *Arm SystemReady IR ACS Tests* for more details.

Build and Automated Validation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build and run the Arm SystemReady IR ACS tests:

1. Select `Arm SystemReady IR Architecture Compliance Suite (ACS) Tests` under `Arm SystemReady IR Validation` from the `Use-Case` menu.
2. Select `Build`.

A similar output to the following is printed out:

```
NOTE: recipe arm-systemready-ir-acs-2.1.0-r0: task do_testimage: Started
Creating terminal default on terminal_ns_uart0
Creating terminal tf-a on terminal_sec_uart
Creating terminal scp on terminal_uart_scp
Creating terminal lcp on terminal_uart_lcp
Creating terminal rss on terminal_rss_uart
Creating terminal safety_island_c0 on terminal_uart_si_cluster0
```

(continues on next page)

(continued from previous page)

```
Creating terminal safety_island_c1 on terminal_uart_si_cluster1
Creating terminal safety_island_c2 on terminal_uart_si_cluster2
Test Group (PlatformSpecificElements): FAILED
Test Group (RequiredElements): FAILED
Test Group (CheckEvent_Conf): PASSED
Test Group (CheckEvent_Func): PASSED
Test Group (CloseEvent_Func): PASSED
Test Group (CreateEventEx_Conf): PASSED
Test Group (CreateEventEx_Func): PASSED
Test Group (CreateEvent_Conf): PASSED
Test Group (CreateEvent_Func): PASSED
Test Group (RaiseTPL_Func): PASSED
Test Group (RestoreTPL_Func): PASSED
Test Group (SetTimer_Conf): PASSED
Test Group (SetTimer_Func): PASSED
Test Group (SignalEvent_Func): PASSED
Test Group (WaitForEvent_Conf): PASSED
Test Group (WaitForEvent_Func): PASSED
Test Group (AllocatePages_Conf): PASSED
Test Group (AllocatePages_Func): PASSED
Test Group (AllocatePool_Conf): PASSED
Test Group (AllocatePool_Func): PASSED
Test Group (FreePages_Conf): PASSED
Test Group (FreePages_Func): PASSED
Test Group (GetMemoryMap_Conf): PASSED
Test Group (GetMemoryMap_Func): PASSED
...
...
Test Group (virtio_blk virtio1): vda
Test Group (Supported ports):
Linux tests complete
RESULTS:
RESULTS - arm_systemready_ir_acs.SystemReadyACSTest.test_acs: PASSED (32417.37s)
SUMMARY:
arm-systemready-ir-acs () - Ran 1 test in 32417.375s
arm-systemready-ir-acs - OK - All required tests passed (successes=1, skipped=0,␣
↪failures=0, errors=0)
```

As seen in the above logs, some Test Groups are expected to fail. The following messages are expected to validate this Use-Case:

```
RESULTS - arm_systemready_ir_acs.SystemReadyACSTest.test_acs: PASSED (32417.37s)
```

Note: Running the ACS tests more than once will have them resume from where they last stopped. Additionally, consecutive runs are not supported by the ACS logs; it will result in a failure after the end of the tests. Use the following to re-start the entire test suite properly:

```
kas shell -c "bitbake arm-systemready-ir-acs -C unpack"
```

Note: The ACS tests take hours to complete. The actual time taken will vary depending on the performance of the

build host. The default timeout setting for the tests is 12 hours for an x86_64 host or 24 hours for an aarch64 host. If a timeout failure occurs, increase the timeout setting and re-run the tests with the following command on the build host terminal. The example command below changes the timeout setting to 16 hours.

```
TEST_OVERALL_TIMEOUT="\${@16*60*60}" kas shell -c "bitbake arm-systemready-ir-acs -C␣
↪unpack"
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

Refer to *Arm SystemReady IR ACS Tests* for an explanation on how the ACS tests are set up and how they work in the Reference Software Stack.

Linux Distribution Installation (Debian and openSUSE)

The Arm SystemReady IR-aligned firmware must boot at least two unmodified generic UEFI distribution images from an ISO image.

This Software Stack currently supports two Linux distributions: Debian Stable and openSUSE Leap.

Note: The installation of a Linux distribution requires some manual interaction, for example, some necessary selections or confirmations, entering the user and password, etc.

The whole installation process takes a long time (possibly up to 10 hours, or even longer).

We suggest that when running the Linux distribution installations the FVP is the only running process as it will consume large amounts of RAM that can make the system unstable.

Refer to *Linux Distributions Installation Tests* for an explanation on how the Linux distros installation is set up and how they work in the Reference Software Stack.

Debian

To install Debian, you can refer to the Debian GNU/Linux Installation Guide.

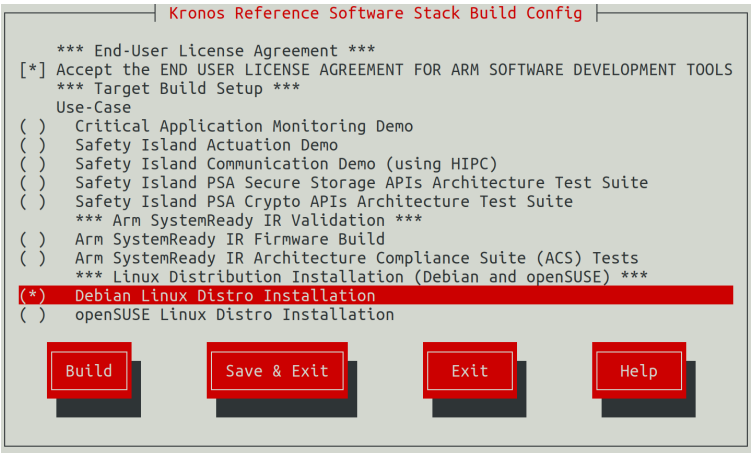
Distro Installation Media Preparation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build the Arm SystemReady IR Linux distros installation tests:

1. Select `Debian Linux Distro Installation` under `Linux Distribution Installation (Debian and openSUSE)` from the `Use-Case` menu.
2. Select `Build`.

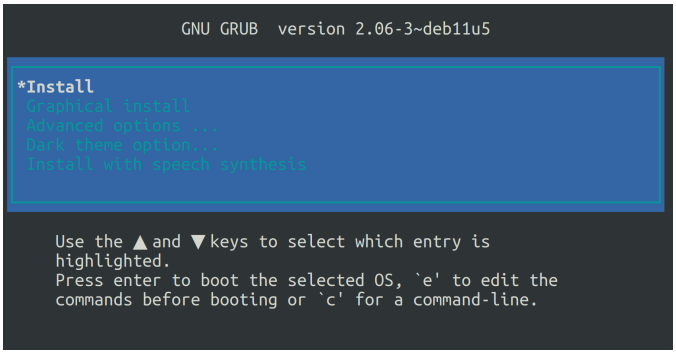


Distro Installation

Run the following command to start the installation:

```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The whole process of installing Debian will probably take about 5 hours. The install process begins when you see the following:

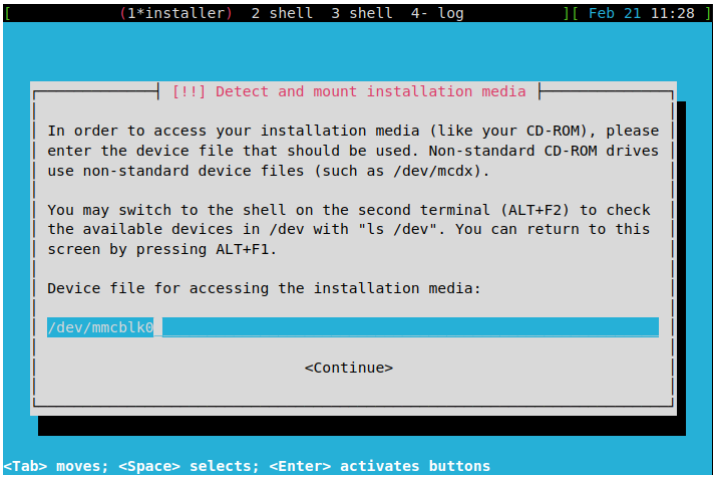


Select `Install` to start the installation process.

The following are problems that have been encountered during the Debian installation process and how to solve them:

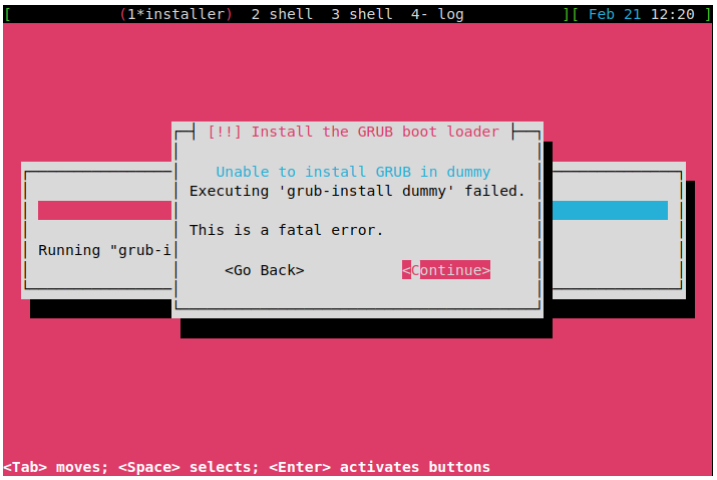
- Detect and mount installation media

    1. After the installer starts, a tab titled `Detect and mount installation media` will appear with `No device for installation media was detected.` When prompted with `Load drivers from removable media?` select `No` to continue.
    2. For `Manually select a module and device for installation media?` select `Yes`.
    3. For `Module needed for accessing the installation media:` select `none`.
    4. For `Device file for accessing the installation media:` input `/dev/mmcblk0` as the device file for accessing the installation media, then select `Continue`.



- Install the GRUB boot loader

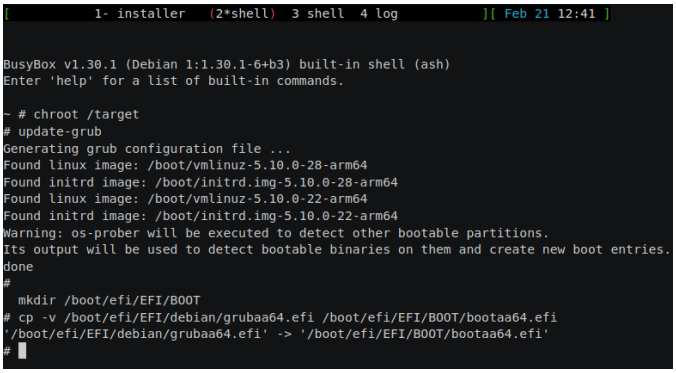
  When the installation reaches the `Install the GRUB boot loader` phase, there will be an error `Unable to install GRUB in dummy`. This is because on an EBBR platform, UEFI `SetVariable()` is not required at runtime (however, it is required at boot time).



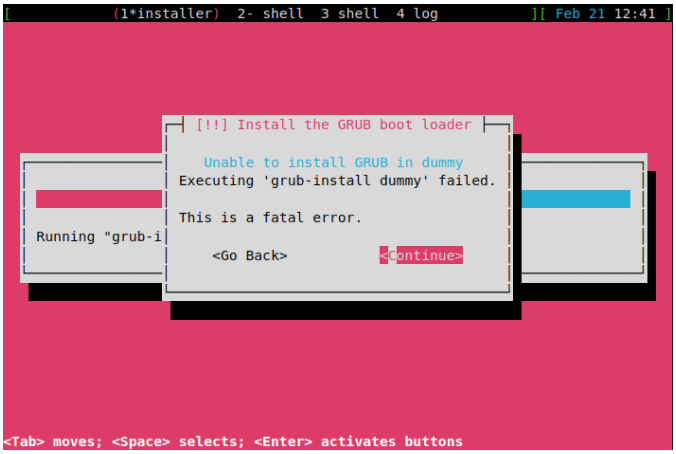
  One workaround we have is to "execute a shell" when the GRUB install phase throws the above error. To execute a shell, press `Ctrl-a n` to switch the debug shell, and run the following commands:

```
chroot /target
update-grub
mkdir /boot/efi/EFI/BOOT
cp -v /boot/efi/EFI/debian/grubaa64.efi /boot/efi/EFI/BOOT/bootaa64.efi
```

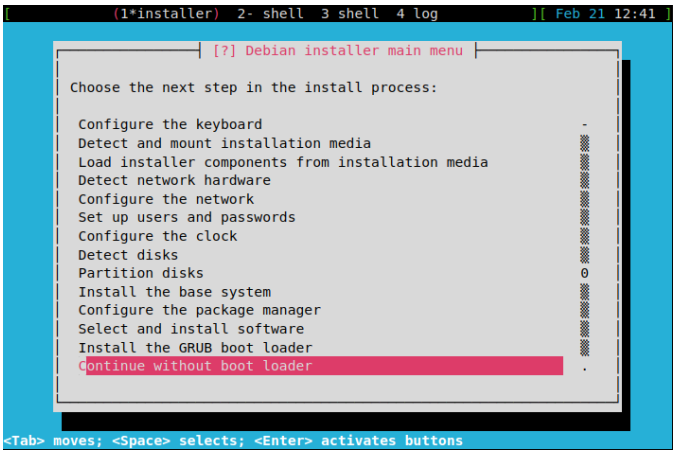
  A snapshot is as below:



After doing the above GRUB workaround, press `Ctrl-a p` to go back to the installer again. Select `Continue` on the GRUB failure screen.



Select `Continue without boot loader` in the `Debian installer main menu` and continue.



- Log in

  When the installation reaches the final `Finishing the installation` phase, you will need to wait some time to finish the remaining tasks, and then it will automatically reboot into the installed OS. You can log into the Linux shell with the user created during installation.

- Terminate the FVP

  To shut down the FVP and terminate the emulation automatically, log into the Linux shell as the root user then run the following command.

  ```
  shutdown now
  ```

  The below message indicates the shutdown process is complete.

  ```
  reboot: Power down
  ```

  Subsequently running the FVP will boot into Debian.

openSUSE

To install openSUSE, you can refer to the openSUSE Installation Guide.

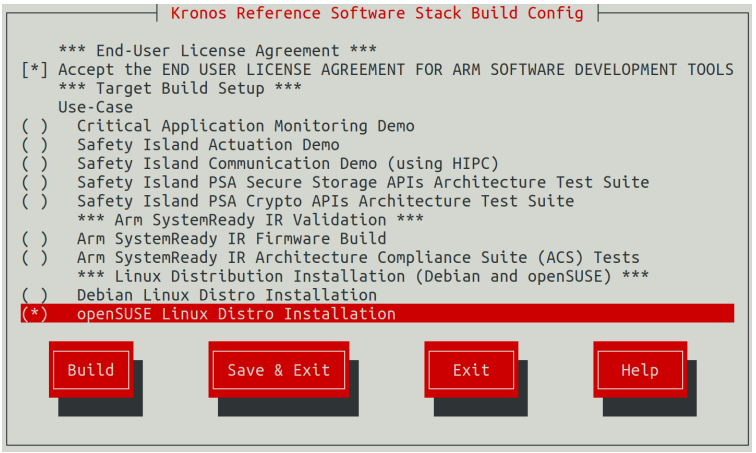
Distro Installation Media Preparation

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build the Arm SystemReady IR Linux distros installation tests:

1. Select `openSUSE Linux Distro Installation` under `Linux Distribution Installation (Debian and openSUSE)` from the `Use-Case` menu.
2. Select `Build`.



Distro Installation

Run the following command to start the installation:

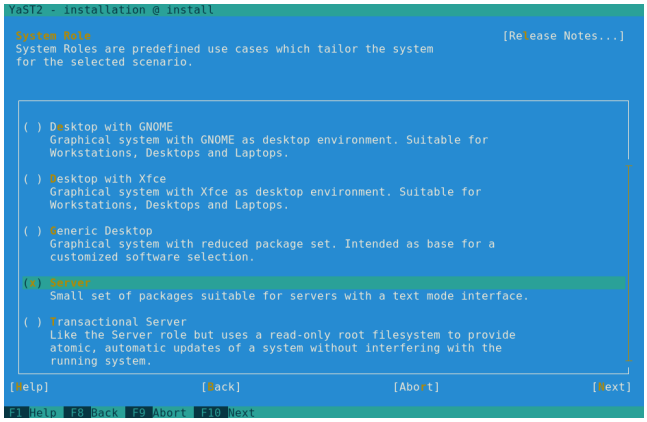
```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

The whole process of installing openSUSE will take several hours. The install process begins when you see the following:

Select `Installation` to start the installation process.

- System Role

  When you get to the `System Role` screen, select `Server`, then select `Next` to continue with the installation.



> **Tip:** Use `Tab` to cycle through options on screens during installation.

- Installation process

  Once you have selected `Install` on the `Confirm Installation` screen, the installation will proceed and it will take several hours. The steps of the installation process are:

  - `Installing Packages...`

  - Save configuration
  - Save installation settings
  - Install boot manager
  - Prepare system for initial boot
  - Then the system will reboot automatically in 10s, you can select OK to reboot immediately.

- Log in

  After the reboot process, log into the Linux shell with the user created during installation.

- Terminate the FVP

  To shut down the FVP and terminate the emulation automatically, run the following command.

```
sudo shutdown now
```

  The below message indicates the shutdown process is complete.

```
reboot: Power down
```

  Subsequently running the FVP will boot into openSUSE.

Secure Firmware Update

Currently, *Secure Firmware Update* is only available in the Baremetal Architecture.

Baremetal Architecture

Build

The to be updated firmware capsule for testing will be generated together with the image for the software stack when building. The firmware capsule is placed on a removable storage device (in the case of Kronos, an MMC card implementation in the FVP).

Ensure the creation of the initial firmware flash images because previously updated firmware will lead to failure of the secure firmware update tests.

```
kas shell -c "bitbake ap-flash-image rss-flash-image -C image"
```

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To build a Baremetal Architecture image:

1. Select `Critical Application Monitoring Demo` from the `Use-Case` menu.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Build`.

Run the FVP

To start the FVP and connect to the Primary Compute terminal (running Linux):

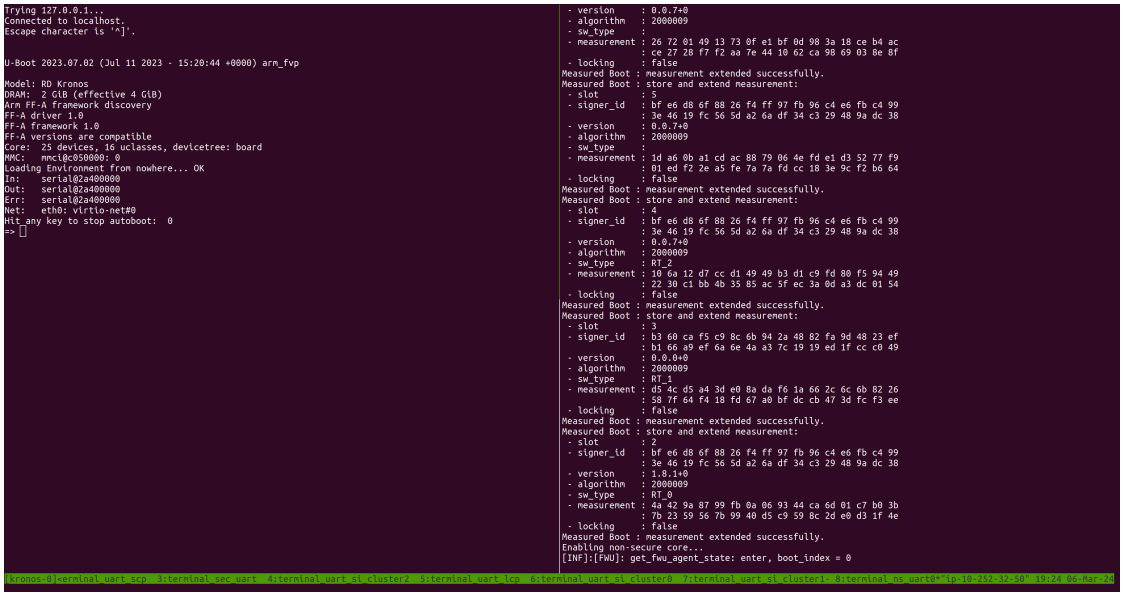
```
kas shell -c "../layers/meta-arm/scripts/runfvp -t tmux --verbose"
```

Note that the main tmux windows involved in the Secure Firmware Update are `terminal_ns_uart0` and `terminal_rss_uart`. For ease of navigation, it is recommended to join these in a single window with two panes.

Follow the steps below to achieve the same:

1. Ensure that the tmux window titled `terminal_ns_uart0` is selected. If not, press `Ctrl-b w` from the tmux session, navigate to the tmux window titled `terminal_ns_uart0` followed by pressing the `Enter` key.
2. The user should wait for the U-Boot `Hit any key to stop autoboot` to appear.
3. Press any key before the time limit to enter the U-Boot shell.
4. Press `Ctrl-b :` and then type `join-pane -s :terminal_rss_uart -h` followed by pressing the `Enter` key to join the RSS terminal window to the Primary Compute terminal window.

Refer to the following image of the tmux panes rearrangement. Panes can be navigated using `Ctrl-b` followed by the arrow keys.



Run the Demo

To start Secure Firmware Update:

1. In the U-Boot shell, run the following commands to start Secure Firmware Update:

   Note: Each command should be copied and pasted individually to the U-Boot shell.

   ```
   fatload mmc 0:1 0xa2000000 fw.cap
   efidebug capsule update -v 0xa2000000
   ```

2. The system will automatically start upgrading the firmware capsule. **Note: This time there is no need to press any keys.**

   The following logs indicate that the upgrade process has started and is in progress.

   In `terminal_ns_uart0`:

   ```
   FF-A driver 1.0
   FF-A framework 1.0
   FF-A versions are compatible
   EFI: MM partition ID 0x8003
   EFI: FVP: Capsule shared buffer at 0x81000000 , size 8192 pages
   ```

   In `terminal_rss_uart`:

   ```
   [INF]:[FWU]: get_fwu_agent_state: enter, boot_index = 0
   [INF]:[FWU]: get_fwu_agent_state: enter, boot_index = 0
   [INF]:[FWU]: FMP image update: image id = 1
   [INF]:[FWU]: FMP image update: status = 0, version=7, last_attempt_version=0.
   [INF]: [FWU]: Host acknowledged.
   [INF]:[FWU]: pack_image_info:207 ImageInfo size = 105, ImageName size = 14,␣
   ↪ImageVersionName size = 14
   [INF]: [FWU]: Getting image info succeeded.
   [INF]:[FWU]: get_fwu_agent_state: enter, boot_index = 0
   [INF]:[FWU]: uefi_capsule_retrieve_images: enter, capsule ptr = 0x0x65000000
   [INF]:[FWU]: uefi_capsule_retrieve_images: capsule size = 18284656, image count = 1
   [INF]:[FWU]: uefi_capsule_retrieve_images: image 0, version = 3
   [INF]:[FWU]: uefi_capsule_retrieve_images: image 0 at 0x65000070, size=18284560
   [INF]:[FWU]: flash_rss_capsule: enter: image = 0x65000070, size = 16187408, version␣
   ↪= 3
   [INF]:[FWU]: erase_bank: erasing sectors = 4080, from offset = 16748544
   [INF]:[FWU]: flash_rss_capsule: writing capsule to the flash at offset = 16748544...
   ```

   Note: This step will take about 10 minutes.

3. The system will reset after a successful firmware update and boot with the updated firmware. This can be confirmed by checking the terminal logs; if there are lines in the log like below, then the upgrade was successful and the system has successfully rebooted with the updated firmware.

   In `terminal_rss_uart`:

   ```
   [INF]: [FWU]: Flashing the image succeeded.
   [INF]: [FWU]: Performing system reset...
   ...
   ```

(continues on next page)

(continued from previous page)

```
...
[INF]:[FWU]: get_fwu_agent_state: enter, boot_index = 1
```

4. The system will eventually boot into Linux using the upgraded firmware.
5. To shut down the FVP and terminate the emulation, select the terminal titled as `python3` where the `runfvp` was launched by pressing `Ctrl-b 0` and press `Ctrl-c` to stop the FVP process.

Automated Validation

Ensure the creation of the initial firmware flash images because previously updated firmware will lead to failure of the tests.

```
kas shell -c "bitbake ap-flash-image rss-flash-image -C image"
```

To run the configuration menu:

```
kas menu kronos/Kconfig
```

To run the validation tests:

1. Select `Critical Application Monitoring Demo` as `Use-Case`.
2. Select `Baremetal` from the `Reference Software Stack Architecture` menu.
3. Select `Run Automated Validation` from the `Runtime Validation Setup` menu.
4. Select `Build`.

The following messages are expected in the output to validate this Use-Case:

```
RESULTS - test_00_fwu.SecureFirmwareUpdateTest.test_securefirmwareupdate: PASSED (414.
↪85s)
```

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for possible workarounds.

See *Integration Tests Validating Secure Firmware Update* for more details.

Describes how to reproduce a Reference Software Stack image, and how to configure, build, run and validate the supported set of architecture features and *Use-Cases*.

2.2 Borrow

To reuse the components and patches of the Reference Software Stack, refer to each of the individual components mentioned in *Components*.

2.2.1 Downstream Changes

Detailed below is a table linking each component of the Reference Software Stack to its Downstream Changes section. Each section contains the component's patch files and a high level overview of their purpose/functionality as a group:

Component	Link to the Downstream Changes
RSS	*Downstream Changes* for RSS
SCP-firmware	*Downstream Changes* for SCP-firmware
Trusted Firmware-A	*Downstream Changes* for TF-A
OP-TEE	*Downstream Changes* for OP-TEE
Trusted Services	*Downstream Changes* for Trusted Services
U-Boot	*Downstream Changes* for U-Boot
Xen	*Downstream Changes* for Xen
Linux Kernel	*Downstream Changes* for Linux
Zephyr	*Downstream Changes* for Zephyr

Provides an overview of the key components and the applied changes for this project so that the user can determine how to isolate and build them independently and import them into their existing project.

CHAPTER

THREE

SOLUTION DESIGN

3.1 Boot Process

3.1.1 RSS-oriented Boot Flow

The *RSS* is the root of trust chain. It is the first booting element when the system is powered up.

The boot sequence is shown in the RSS-oriented Boot Flow diagram below *Boot Flow* section.

The RSS uses a NVM flash to store the images of various components, including:

- RSS BL2 image
- RSS Runtime image
- SCP RAM Firmware (SCP RAMFW) image
- LCP RAM Firmware (LCP RAMFW) image
- Safety Island Cluster 0 (SI CL0) image
- Safety Island Cluster 1 (SI CL1) image
- Safety Island Cluster 2 (SI CL2) image
- Application Processor BL2 (AP BL2) image

Trust Chain

To make the platform secure, it is critical to protect each booting component from the execution of malicious code. This is implemented by building a trust chain where each step in the execution chain authenticates the next step before execution.

The authentication of the images is done with hash (SHA-256) and digital signature (RSA-3072) validation.

Image Signing

A RSA private key is stored in TF-M's source code repository (the `bl2/ext/mcuboot/root-RSA-3072.pem` file) for testing. The private key is used to sign the images listed above that RSS BL2 loads.

In the Yocto build stage of the Kronos platform, a shell function `sign_host_image()` is used to sign the images, which can be found at meta-arm/classes/tfm_sign_image.bbclass. Then the signed images are written to the NVM flash.

System Provisioning and Image Authentication

A public key is derived from the private key for authenticating the signed images. The public key is also known as the Root of Trust Public Key (ROTPK). It is also written in the NVM flash in the build stage. The hash of the public key is written in the `dm_dummy_provisioning_data.c` file of the TF-M source code folder `platform/ext/target/arm/rss/common/provisioning/bundle_dm/`.

During the system's first boot, the hash of the public key is provisioned into the OTP by BL1_1. More details on the provisioning can be found in the RSS provisioning page. Once the provisioning stage has been completed, the OTP contents cannot be updated.

BL2 reads the public key from the NVM flash and validates the public key against the hash that has been provisioned in the OTP. Then BL2 uses the public key to authenticate the images.

Key Customization

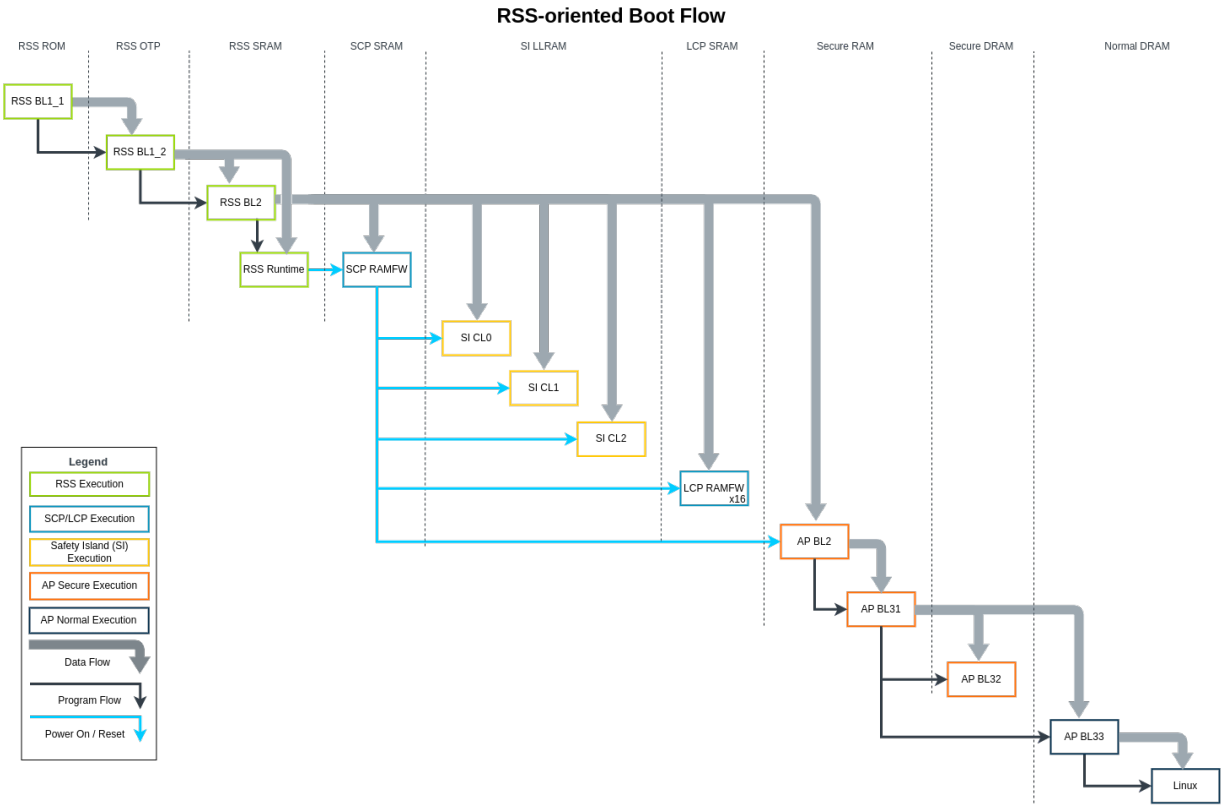
The default private key used in the Kronos platform should only be used for test purposes. Since this private key is widely distributed, it should never be used for production. To replace the default key, the user needs to:

- Generate a new RSA key pair
- Replace the default private key `bl2/ext/mcuboot/root-RSA-3072.pem` with the new private key
- Generate the hash of the public key and replace the definition of `ASSEMBLY_AND_TEST_PROV_DATA_KIND_0` in `dm_dummy_provisioning_data.c` with the hash value.

For detail of how to generate the private key and the hash of the public key, refer to the documentation of imgtool which is provided by MCUboot.

Boot Flow

The following diagram illustrates the boot flow that originates from the RSS.



Major steps of the boot flow:

1. RSS BL1_1:
    - Begins executing in place from ROM when the system is powered up
    - Provisions RSS BL1_2 and various keys and other data from the provisioning bundle to the OTP (This step only happens on the system's first boot)
    - Copies the RSS BL1_2 image from the OTP to the SRAM
    - Validates RSS BL1_2 against the hash stored in the OTP
    - Transfers the execution to RSS BL1_2
2. RSS BL1_2:
    - Copies the encrypted RSS BL2 image from flash into the SRAM
    - Decrypts the RSS BL2 image
    - Transfers the execution to RSS BL2
3. RSS BL2:
    - Copies the SCP RAMFW image from flash to SCP SRAM and authenticates the image
    - Releases the SCP out of reset

- Copies the SI CL0 image from flash to SI LLRAM and authenticates the image
- Notifies the SCP to power on the SI CL0
- Copies the SI CL1 image from flash to SI LLRAM and authenticates the image
- Notifies the SCP to power on the SI CL1
- Copies the SI CL2 image from flash to SI LLRAM and authenticates the image
- Notifies the SCP to power on the SI CL2
- Copies the LCP image from flash to LCP SRAM and authenticates the image
- Releases the LCP out of reset
- Copies the AP BL2 image from flash to AP SRAM and authenticates the image
- Notifies the SCP to power on the AP

3.1.2 Primary Compute Boot Flow

The Application Processor (AP) refers to the cores in the Primary Compute of the Kronos Reference Design. The purpose of its firmware is to provide an Arm SystemReady IR-aligned interface to Linux. Arm SystemReady IR compatible systems are required to follow the Device Tree specification, so the *U-Boot* bootloader is used in the Normal world, which provides the UEFI implementation and exposes the device tree to Linux.

Trusted Firmware-A provides the initial, Secure world firmware, which consists of BL2 and BL31. BL32 is provided by OP-TEE. BL33 is provided by U-Boot.

The Primary Compute uses:

- Secure Flash to store the following components:
  - AP BL31
  - AP BL32 (OP-TEE)
  - AP BL33 (U-Boot)
- First VFAT (boot) partition of the VirtIO Block to store the following components:
  - GRUB2
  - Linux (Baremetal Architecture)
  - Xen (Virtualization Architecture)

The Primary Compute boot flow follows the following steps:

1. AP BL2:
   - Copies the AP BL31 image from Secure Flash to Secure RAM
   - Transfers the execution to AP BL31
2. AP BL31:
   - Copies the AP BL32 (OP-TEE) image from Secure Flash to Secure DRAM
   - Transfers the execution to AP BL32
   - Copies the AP BL33 (U-Boot) image from Secure Flash to Normal DRAM
   - Transfers the execution to AP BL33
3. AP BL33 loads GRUB2 from the boot partition

4. Grub loads and boots either Linux (Baremetal Architecture) or Xen (Virtualization Architecture) from the boot partition, depending on the Grub configuration

3.2 Secure Services

3.2.1 Introduction

The Reference Software Stack provides the implementation of Secure Services through both the Primary Compute and Safety Island. These services are aligned to the following specifications:

- PSA Crypto API: The API provides a portable programming interface to cryptographic operations, and key storage functionality on a wide range of hardware.
- PSA Secure Storage API: The API provides key/value storage interfaces for use with device-protected storage. The Secure Storage API describes two interfaces for storage:
  - Internal Trusted Storage (ITS) API: An interface for storage provided by the Platform Root of Trust (PRoT). For now the ITS API is not supported by the Reference Software Stack on the Primary Compute.
  - Protected Storage (PS) API: An interface for external protected storage.

3.2.2 Primary Compute Secure Services

On Primary Compute, the implementation of Crypto Service and Secure Storage Service is based on the SE Proxy secure partition.

The Primary Compute also provides the implementation of UEFI SMM Services via the SMM Gateway secure partition to support UEFI System Management Mode (SMM).

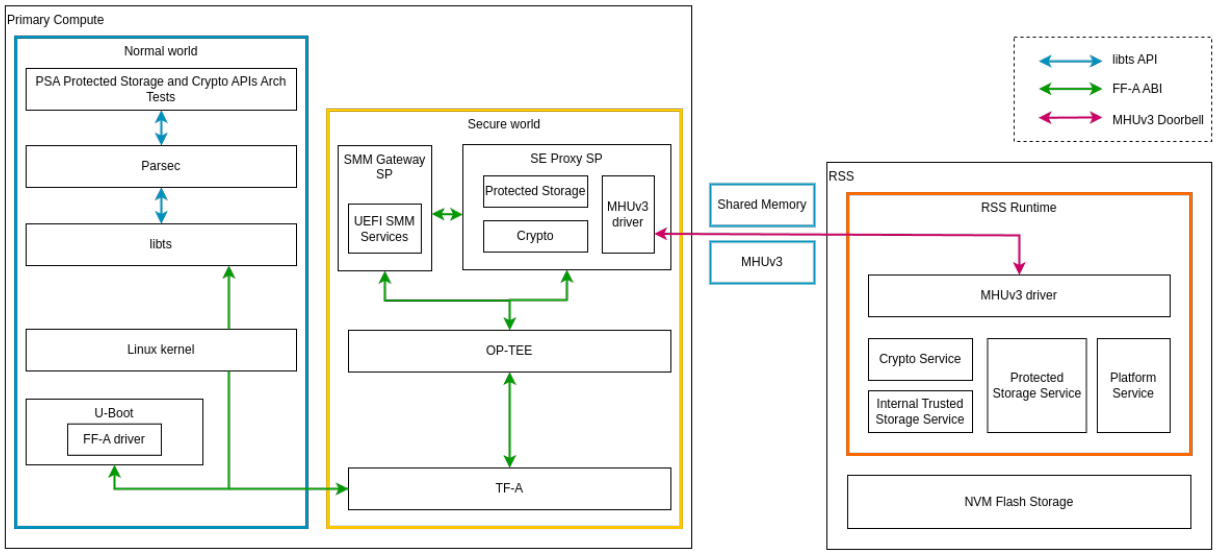
These Secure Services are provided by the Trusted Services project, and implemented by leveraging the TrustZone technology in the Primary Compute and the hardware-isolated secure enclave in the RSS.

The Reference Software Stack provides the implementation of Secure Services through both the Primary Compute and the Safety Island.

Architecture

The following diagram illustrates the components and data flow that implement the Primary Compute Secure Services.

Primary Compute Secure Services



PSA Protected Storage and Crypto APIs Arch Tests

The PSA Protected Storage and PSA Crypto APIs Arch Tests can be accessed from the Primary Compute linux terminal by running a single command for each. The test suites execute over around a minute, and a table of results is displayed upon completion.

Refer to *Integration Tests Validating Primary Compute PSA APIs Architecture Test Suite* for more information.

Parsec

Parsec, The Platform AbstRaction for SECurity, is an open-source initiative to provide a common API to hardware security and cryptographic services in a platform-agnostic way. This abstraction layer keeps workloads decoupled from physical platform details.

`Parsec` is configured to use Trusted Services in the Secure world as its backend. `Parsec` service calls the API provided by `libts` which further invokes the RSS for cryptographic services.

libts

In Linux userspace, the Secure Services are provided in the form of libts API. `libts` is a library that is provided by Trusted Services for handling service discovery and Remote Procedure Call (RPC) messaging. `libts` entirely decouples client applications from details of where a service provider is deployed and how to communicate with it.

The client application sends operation requests and receives responses by calling the `libts` API. `libts` communicates with the Secure Partition (SP) running in the Secure world. The communication between `libts` and the Secure world SP is carried by the Arm Firmware Framework for Arm A-profile (FF-A) call which is supported by Linux kernel and Trusted Firmware-A.

SE Proxy SP

The SE Proxy SP (Secure Enclave Proxy Secure Partition) is a proxy partition managed by OP-TEE. It provides access to services hosted by the RSS.

The `SE Proxy SP` receives secure service operation requests from the Normal world, translates the request parameters to IPC calls, and invokes the runtime services provided by the RSS. The IPC is carried by Shared Memory and MHUv3 Doorbell communication between the Primary Compute and the RSS.

SMM Gateway SP

The SMM Gateway SP (System Management Mode Gateway Secure Partition) serves as a gateway for the variable storage required by the implementation of UEFI Boot and Runtime Services APIs. These UEFI variables are stored in the Protected Storage Service provided by the RSS.

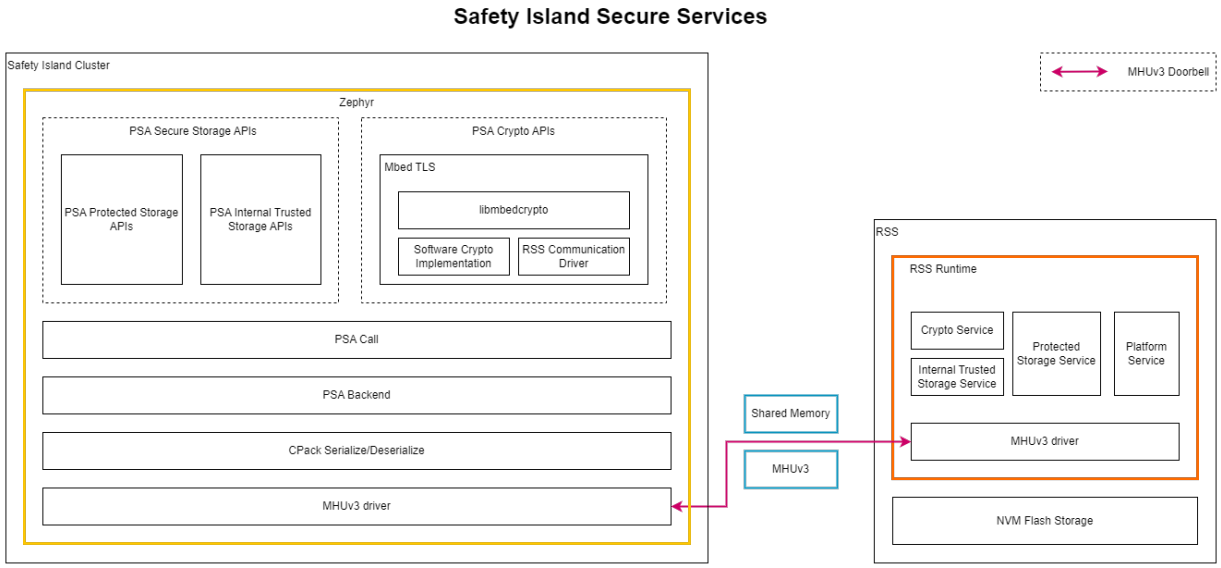
The data flow to store UEFI variables is presented in the diagram at the beginning of the *Architecture* section. The U-Boot implementation of the UEFI subsystem uses the FF-A driver to communicate with the UEFI SMM Services in the SMM Gateway SP. The backend of the SMM services uses the Protected Storage proxy from the SE Proxy SP. From there on, the Protected Storage calls are forwarded to the secure enclave as explained above.

3.2.3 Safety Island Secure Services

The Safety Island provides the implementation of Crypto Service and Secure Storage Service. The data paths of the services are different.

Architecture

The following diagram illustrates the components and data flow that implement the Safety Island Secure Services.



PSA Crypto APIs

The PSA Crypto API is implemented by the `libmbedcrypto` library of Mbed TLS.

Mbed TLS supports drivers for cryptographic accelerators, secure elements and random generators. An *RSS Communication Driver* is created to communicate with RSS for calling the crypto service that is provided there. The driver invokes the `psa_call()` interface to communicate with the RSS via MHUv3.

By introducing the driver, different crypto operations can be handled in different ways:

- Asymmetric crypto operations can be handled in RSS for enhanced security, because the private key cannot leave RSS. The following Crypto APIs are supported by the driver:

  Key management:

  - `psa_import_key`
  - `psa_generate_key`
  - `psa_copy_key`
  - `psa_destroy_key`
  - `psa_export_key`
  - `psa_export_public_key`

  Asymmetric signature:

  - `psa_sign_message`
  - `psa_verify_message`
  - `psa_sign_hash`
  - `psa_verify_hash`

  Asymmetric encryption:

  - `psa_asymmetric_encrypt`
  - `psa_asymmetric_decrypt`

- Symmetric and other crypto operations are handled in Safety Island locally with the Mbed TLS software implementation, where the runtime performance is optimized.

PSA Secure Storage APIs

Two use cases are addressed by PSA Secure Storage API:

- Internal Trusted Storage: Internal Trusted Storage aims at providing a place for devices to store their most intimate secrets, either to ensure data privacy or data integrity. For example, a device identity key requires confidentiality, whereas an authority public key is public data but requires integrity. Other critical values that are part of a Root of Trust Service — for example, secure time values, monotonic counter values, or firmware image hashes — will also need trusted storage.

  The following PSA Internal Trusted Storage APIs are supported in Kronos Reference Software Stack:

  - `psa_its_set`
  - `psa_its_get`

  - `psa_its_get_info`
  - `psa_its_remove`

- Protected Storage: Protected Storage is meant to protect larger data-sets against physical attacks. It aims to provide the ability for a firmware developer to store data onto external flash, with a promise of data-at-rest protection, including device-bound encryption, integrity, and replay protection. It should be possible to select the appropriate protection level — for example, encryption only, or integrity only, or both — depending on the threat model of the device and the nature of its deployment.

  The following PSA Protected Storage APIs are supported in Kronos Reference Software Stack:

  - `psa_ps_set`
  - `psa_ps_get`
  - `psa_ps_get_info`
  - `psa_ps_remove`
  - `psa_ps_get_support`

All the PSA Secure Storage API interfaces use the `psa_call()` for communicating with the RSS.

The PSA APIs are thread safe in case of parallel API invocations from multiple threads within the same cluster or from different clusters, Where the `psa_call()` blocks any new requests using a semaphore until the ongoing request completes.

Memory Map

RSS shares dedicated SRAM with Safety Island Clusters 0, 1, and 2 and Primary Compute.

Safety Island side:

Cluster 0:

- `local_sram_rss_cl0` : Used for data transfer between Cluster 0 and RSS

  Refer to the device tree overlay below for more information about the memory addresses and region sizes.

  - components/safety_island/zephyr/src/overlays/psa/fvp_rd_kronos_safety_island_c0.overlay.

Cluster 1:

- `local_sram_rss_cl1` : Used for data transfer between Cluster 1 and RSS

  Refer to the device tree overlay below for more information about the memory addresses and region sizes.

  - components/safety_island/zephyr/src/overlays/psa/fvp_rd_kronos_safety_island_c1.overlay.

Cluster 2:

- `local_sram_rss_cl2` : Used for data transfer between Cluster 2 and RSS

  Refer to the device tree overlay below for more information about the memory addresses and region sizes.

  - components/safety_island/zephyr/src/overlays/psa/fvp_rd_kronos_safety_island_c2.overlay.

Primary Compute side:

- `rss_comms-virtio` : Used for data transfer between SE Proxy SP in the Primary Compute Secure World and RSS

RSS communication

The RSS communication protocol is designed to be a lightweight serialization of the `psa_call()` API through a combination of in-band MHUv3 (Message Handling Unit) transport and parameter-passing through Shared Memory.

To call an RSS service, the client must send a message in-band over the MHUv3 sender link to RSS and wait for a reply message on the MHUv3 receiver. The messages are defined as packed C structures, which are serialized in byte-order over the MHUv3 links.

3.2.4 RSS Secure Firmware

The Secure Services are finally served by the `RSS Secure Firmware`. For more information about how the Secure Services work in the RSS, read the TF-M Secure Services page.

Trusted Firmware-M has some limitations regarding the Secure Storage Service. Refer to the release notes *Limitations* section for more details.

3.3 Secure Firmware Update

3.3.1 Introduction

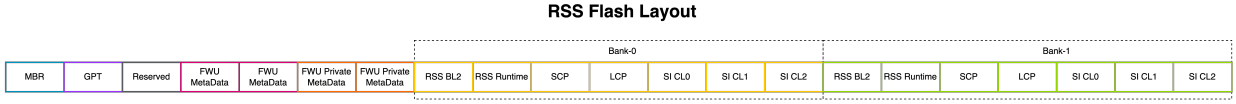
The Reference Software Stack implements Secure Firmware Update following the Platform Security Firmware Update Specification. The following firmware images are included:

- RSS BL2 image
- RSS Runtime image
- SCP RAM Firmware (SCP RAMFW) image
- LCP RAM Firmware (LCP RAMFW) image
- Safety Island Cluster 0 (SI CL0) image
- Safety Island Cluster 1 (SI CL1) image
- Safety Island Cluster 2 (SI CL2) image
- Primary Compute FIP (Firmware Image Package) image

The new images are accepted in the form of a UEFI capsule.

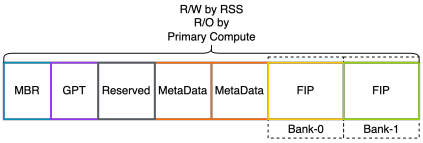
3.3.2 Architecture

As standardized into the Platform Security Firmware Update Specification, each one of the RSS flash and secure flash is divided into two banks, where one bank has the currently running images and the other bank is used for staging new images. The flash layouts are shown in the following figures.



- MBR: Master Boot Record
- GPT: GUUID Partition Table
- FWU MetaData: Used for RSS BL1 to select the correct bank to load and boot the RSS BL2.
- FWU Private MetaData: Used for the RSS BL2 to select the correct bank to load and boot the SCP, LCP, SI, and the Primary Compute BL2.

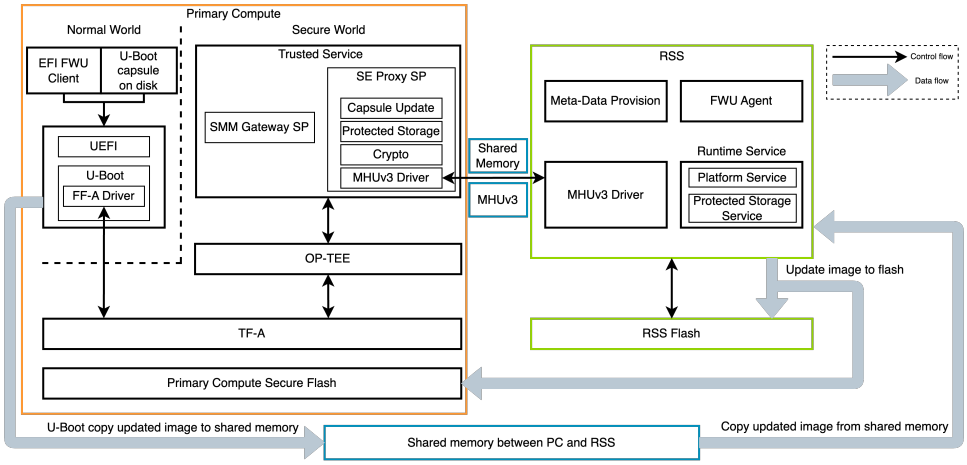
Primary Compute Secure Flash Layout



- MetaData: Used for the Primary Compute BL2 to select the correct bank to load and boot the Primary Compute BL31, BL32, BL33 etc.

The following diagram illustrates the components and data flow that implement the Secure Firmware Update.

Secure Firmware Update Architecture



A typical Secure Firmware Update process can be described in the following steps:

1. The capsule image is generated by EDK2 tools and stored on the disk of the Primary Compute for the UEFI UpdateCapsule runtime service to access.
2. The firmware upgrade process is initiated from the UEFI UpdateCapsule runtime service.
3. The capsule image is then read and copied from the Primary Compute disk to the Shared Memory between the Primary Compute and RSS.
4. The Capsule Update service in SE Proxy SP handles the firmware update request. It then sends a request to the RSS Platform Runtime Service to handle the firmware update request.

5. Once the RSS Platform service receives the firmware update request, it firstly carries out validations of the header of the capsule, the version of the images, and the counter of the images, then copies the image from the Shared Memory to the RSS flash, and finally updates the image to the Bank-0 or the Bank-1 of the RSS flash and Primary Compute Secure Flash.
6. The system will reset after a successful firmware update and boot from the bank with the new firmware images. If the firmware update fails, when the user restarts the system from the UEFI shell the system will boot from the bank with the original firmware images.

3.4 Fault Management

3.4.1 Introduction

The Fault Management subsystem for the Safety Island provides a mechanism for capturing, reporting and collating faults from supported hardware in safety-critical designs.

The subsystem interfaces with the following types of devices:

- A fault device, which reports faults from its safety mechanisms. It may also report faults originating from other fault devices to support the creation of a fault device tree.
- A safety state device, which manages a state in reaction to reported faults.

Supporting driver implementations are provided for the following Arm hardware designs:

- A Device Fault Management Unit (Device FMU): a fault device attached to a GIC-720AE interrupt controller.
- A System Fault Management Unit (System FMU): a fault device which collates faults from upstream FMUs.
- A Safety Status Unit (SSU): a safety state device which manages a state machine in response to faults in a safety-critical system.

Faults

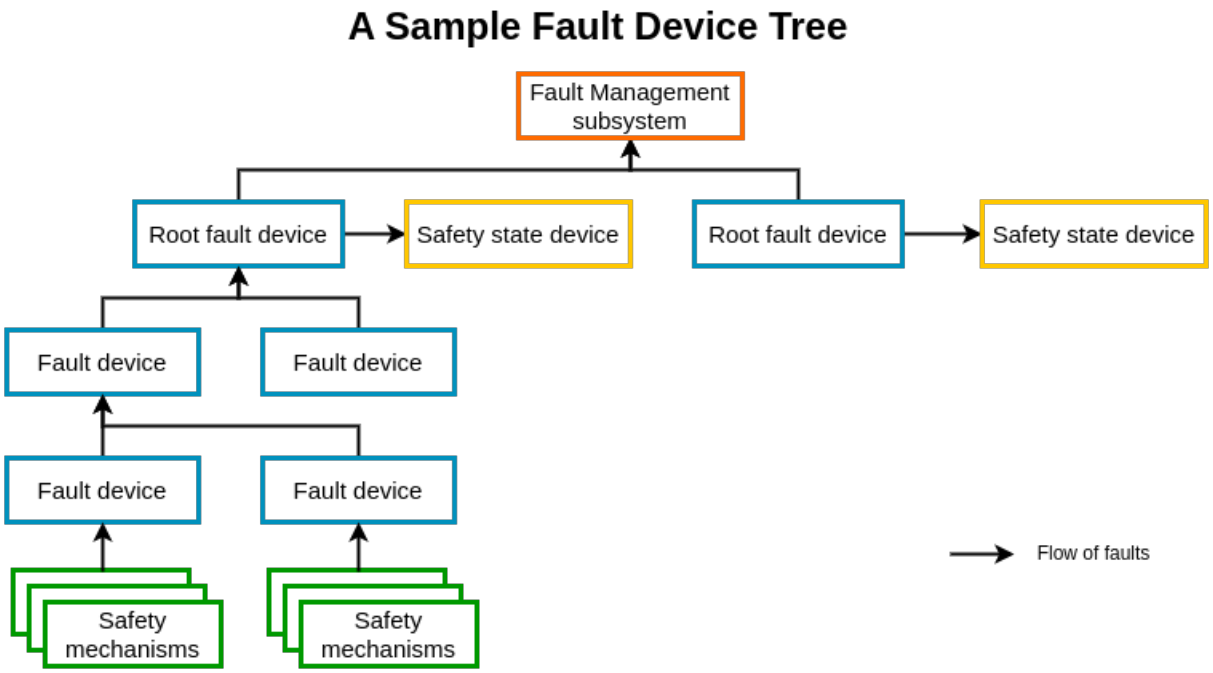
A unique fault (i.e. generated by a specific safety mechanism and reported by a fault device implementation) is represented by the subsystem and driver interfaces as a device-specific 32-bit integer along with a handle to the originating device.

A fault may be critical or non-critical and this affects how it is processed by the subsystem.

Fault Device Trees

The subsystem is configured with a list of "root" fault devices - those located at the root of a fault device tree. Root fault devices are typically collators of faults from multiple upstream fault devices (possibly recursively) and may also directly affect the state of a connected safety state device.

The diagram below shows an illustrative fault device tree. (For the simpler Kronos topology, see *Kronos Deployment* below.)



Safety States

The SSU state machine has 4 safety states:

- `TEST`: Self-test
- `SAFE`: Safe operation
- `ERRN`: Non-critical fault detected
- `ERRC`: Critical fault detected

Control signals from software:

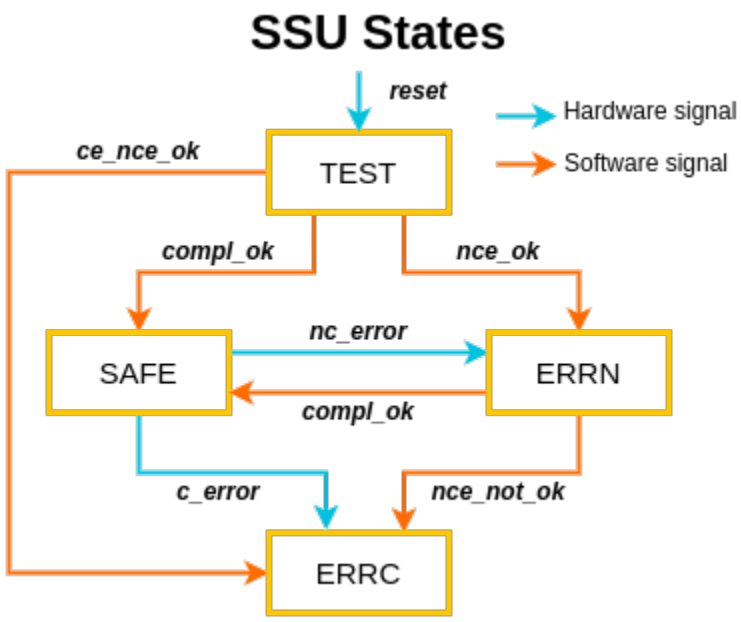
- `compl_ok`: Diagnostics complete or non-critical fault cleared
- `nce_ok`: Non-critical fault diagnosed
- `ce_not_ok`: Critical fault diagnosed
- `nce_not_ok`: Non-correctable non-critical fault

Control signals connected in hardware to the root fault device:

- `nc_error`: Non critical error
- `c_error`: Critical error
- `reset`

`TEST` is the initial state on boot. The software is responsible for transitioning to `SAFE` after the successful completion of a self-test routine. `ERRC` represents a critical system failure, which can only be recovered by resetting the system. A non-critical fault causes a transition to `ERRN`, which can either be recovered back to `SAFE` or promoted to `ERRC` by the software.

The diagram below shows all the possible transitions between these states using these signals.



Finite State Machine (FSM) States and Transitions:

- From reset the FSM defaults to the `TEST` state.
- It shall stay in this state until SW has completed any power up tests. If the SW controlled tests pass then a write can be issued indicating that to move the FSM to the `SAFE` state.
- If the tests fail then a write can be issued to move the FSM to the `ERRN` state, indicating that an error has occurred that may be resolvable.
- After further tests if the SW can issue a write depending on whether it was determined the error has been resolved or not, moving the FSM to `SAFE` it was resolved or `ERRC` if it was not.

  When in `SAFE` mode the FSM can only be moved after either:

  - a reset moving it back to `TEST`
  - a non-critical error interrupt moving it to `ERRN`
  - a critical error interrupt moving it to `ERRC`
  - if a critical and non-critical error occur in the same time the critical error takes precedence and the FSM shall move to `ERRC`

3.4.2 Design

The Fault Management subsystem for the Safety Island implementation and functionality are grounded in the Zephyr real-time operating system (RTOS) environment.

Drivers

Driver interfaces are provided for fault devices and safety state devices. Specific driver implementations with devicetree bindings are provided for the Arm FMU and Arm SSU.

The public driver interfaces are described under components/safety_island/zephyr/src/include/zephyr/drivers/fault_mgmt

The drivers are instantiated in the devicetree using bindings under components/safety_island/zephyr/src/dts/bindings/fault_mgmt

Fault Management Unit

The FMU driver is an implementation of a fault device. Inside the driver, one of two driver implementations is selected at runtime to handle differences between the GIC-720AE and the System FMU programmers' views.

It is expected that interrupts are only defined for root FMUs. If the root FMU is a System FMU, it will collate faults from multiple upstream sources. The driver in this case will inspect the status of other FMUs in the tree when a fault occurs to determine the exact origin and cause of the fault.

The FMU driver allows a single callback to be registered, through which incoming faults are reported.

Safety Status Unit

The SSU driver is an implementation of a safety state device. It implements the safety state device interface which allows its state to be read and controlled.

Subsystem

The Fault Management subsystem manages two fault-handling threads (one for critical faults and another for non-critical faults), which listen for queued faults from any configured root fault device and forward them to all configured fault handlers.

Multiple fault handlers can be statically registered (using the `FAULT_MGMT_HANDLER_DEFINE` macro), each of which is called once per root fault device on initialization, then once per reported fault. Handlers are registered with a unique priority that determines the order in which they are called.

Certain subsystem features are themselves implemented as handlers. It is expected that in order to implement a Fault Management policy for a safety-critical system design, one or more additional custom fault handlers would be required to perform tasks such as:

- Configuring the criticality and enabled state of fault device safety mechanisms.
- Performing a self-test routine before notifying the safety state device that the system is safe for operation.
- Reacting to non-critical faults and deciding whether to perform a corrective action to reset the safety state or promote to a critical fault. This decision may be based on the provided fault count storage.

The subsystem has configuration options to manage the stack space, priority and queue size of both threads, which should be tuned and validated according to deployment requirements. Specifically, more complex custom handlers may require more stack space as they are called on the subsystem threads.

The public interface for the subsystem and its components is described under components/safety_island/zephyr/src/include/zephyr/subsys/fault_mgmt

Safety component

The safety component contains additional interfaces to facilitate reading and updating a system's safety state.

If enabled, this component requires (and validates at boot) that all root fault devices have an attached safety state device.

Storage component

The storage component manages historical counts per safety mechanism per fault device.

Two storage backends are provided:

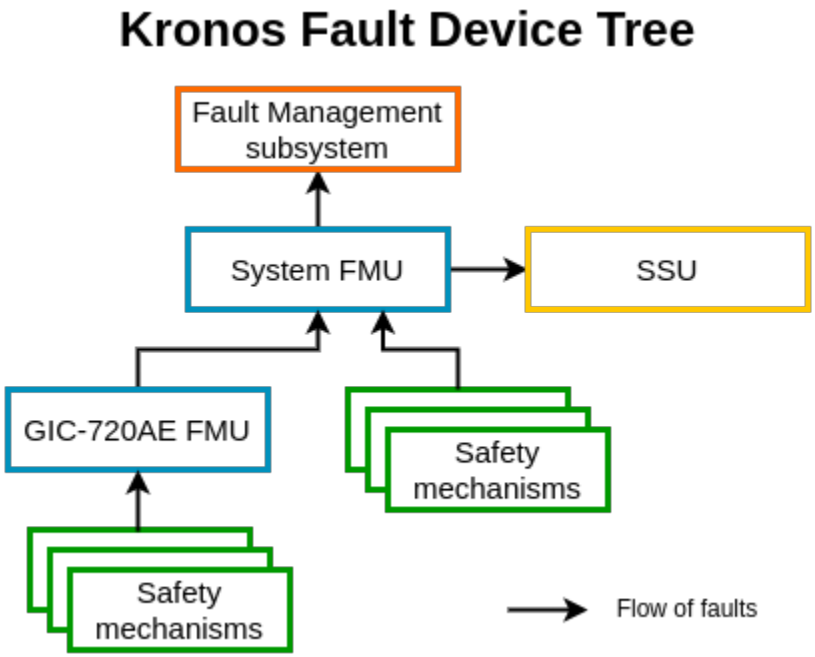
- Trusted Firmware-M PSA Protected Storage Interfaces, with an in-memory cache populated at boot.
- A non-persistent in-memory implementation, using only Zephyr's `sys_hash_map`.

For the PSA backend, there are configuration options to manage the storage key and the maximum record count, which should be tuned and validated depending on the number of distinct faults and devices and/or other system constraints.

3.4.3 Kronos Deployment

The Kronos FVP models:

- An SSU in the Safety Island.
- A System FMU in the Safety Island, attached to the SSU.
- An FMU attached to the GIC-720AE in the Primary Compute, attached to the System FMU.



The Kronos Fault Management application (components/safety_island/zephyr/src/apps/fault_mgmt) provides Kconfig and devicetree overlays for a sample deployment using these devices on Safety Island Cluster 1. The functionality can be evaluated using the Zephyr shell on this cluster. Additionally, this application serves as the basis for the automated validation (see *Integration Tests Validating the Fault Management Subsystem*).

For fault count storage, the application uses the PSA Protected Storage implementation provided by TF-M. `CONFIG_MAX_PSA_PROTECTED_STORAGE_SIZE` is configured according to TF-M storage constraints.

Validation

The Kronos Reference Design contains integration tests for the overall FMU and SSU integration, described at *Integration Tests Validating the Fault Management Subsystem*

3.4.4 Shell Reference

The subsystem provides an optional shell command (enabled using `CONFIG_FAULT_MGMT_SHELL`) which exposes the subsystem API interactively for evaluation and validation purposes. Its sub-commands are described below.

- `fault tree` - Print a description of the fault device tree (including any safety state devices) to the console. The device names printed here can be used in the other commands below.
- `fault inject DEVICE FAULT_ID` - Inject a specific `FAULT_ID` into `DEVICE`. The resultant fault will be logged on the console.
- `fault set_enabled DEVICE FAULT_ID ENABLED` - Enable or disable a specific `FAULT_ID` on a `DEVICE`. Set `ENABLED` to `1` to enable or `0` to disable.
- `fault set_critical DEVICE FAULT_ID CRITICAL` - Configure a specific `FAULT_ID` on a `DEVICE` as critical or non-critical. Set `CRITICAL` to `1` to set as critical or `0` to set as non-critical.

The `FAULT_ID` above refers to a 32-bit integer whose valid values are device-specific (e.g. `0x100` represents an *APB access error* for a System FMU but a *GICD Clock Error* for a GIC-720AE FMU) and opaque to the driver itself.

The following are only available if `CONFIG_FAULT_MGMT_SAFETY` is enabled:

- `fault safety_status DEVICE` - Print the current status of safety state `DEVICE` to the console.
- `fault safety_control DEVICE SIGNAL` - Send `SIGNAL` to safety state `DEVICE`.

The following are only available if `CONFIG_FAULT_MGMT_STORAGE` is enabled:

- `fault list [THRESHOLD]` - List all reported fault counts. The optional `THRESHOLD` filters out faults below a certain count.
- `fault summary` - Show a more detailed summary of the fault counts, including a list of the most reported faults.
- `fault count` - Print the total count of reported faults.
- `fault clear` - Reset all fault counts back to zero.

The test suite at yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_fault_mgmt.py demonstrates usage of these sub-commands.

3.4.5 Safety Considerations

The Fault Management subsystem has the following features to mitigate the risks of unexpected runtime behavior causing a denial of service:

- Iterative methods that take a fixed amount of stack space based on `CONFIG_FAULT_MGMT_MAX_TREE_DEPTH` are used to traverse fault device trees.
- Invalid combinations of configuration values (e.g. a root FMU without IRQ numbers) are detected at compile time where possible.
- The subsystem functionality is composed of independent handlers which can be disabled if not required.

Note that there are conditions where the subsystem will panic and the application running on the Safety Island cluster will stop processing further faults (non-exhaustive):

- Faults arrive more quickly than they are handled over a long enough period for a queue to fill up.
- A fault arrives at a System FMU from an unknown Device FMU.
- The number of stored fault records exceeds the amount of available storage.
- An unexpected error code is returned when attempting to write a fault count to the storage.

3.5 Heterogeneous Inter-Processor Communication (HIPC)

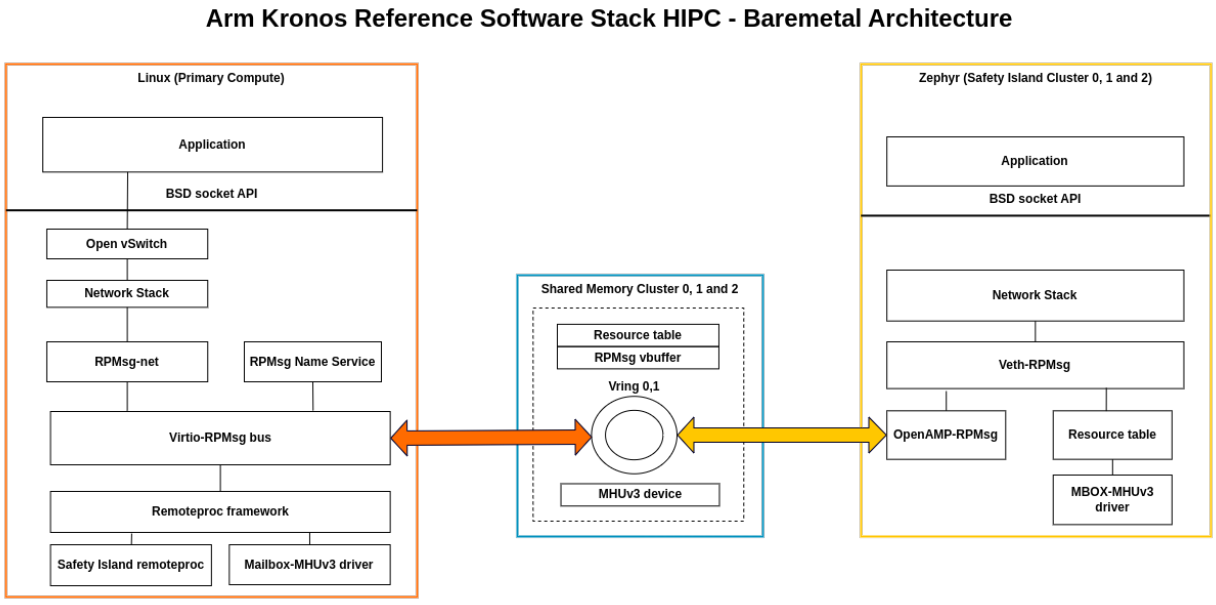
3.5.1 Introduction

The Kronos FVP contains Armv9-A (Primary Compute) and Armv8-R64 (Safety Island) heterogeneous processing elements which share data via the Message Handling Unit (MHUv3) and shared Static Random Access Memory (SRAM). The MHUv3 is a mailbox controller used for signal transmission and the shared memory is used for data exchange. Safety Island clusters also share data via the MHUv3 and shared SRAM.

The HIPC demonstrates the communication between:

- Primary Compute and the three Safety Island clusters.
- Safety Island clusters.

3.5.2 Communication between Primary Compute and Safety Island clusters



RPMsg Protocol

RPMsg (Remote Processor Messaging) is a messaging protocol enabling heterogeneous communication, which can be used by Linux as well as Real Time Operating Systems.

In Linux, the RPMsg framework is implemented on top of the Virtio-RPMsg bus and Remoteproc framework. The Virtio-RPMsg implementation is generic and based on Virtio Vring to transmit/receive messages to/from the remote CPU over shared memory.

On the Safety Island side, Zephyr has imported OpenAMP as an external module. The OpenAMP library implements the RPMsg backend based on Virtio, which is compatible with the upstream Linux Remoteproc and RPMsg components. This library can be used with the Zephyr kernel or Zephyr applications to behave as an RPMsg backend service for communication with the Primary Compute.

Virtual Network Device over RPMsg

RPMsg offers a range of user APIs for RPMsg endpoints to send and receive messages to and from these endpoints. These APIs are suitable for simple inter-processor communication. However, many current user applications are not built on RPMsg APIs. Instead, they use BSD sockets for IPC. The reason for this is that BSD sockets can abstract the difference between inter-processor communication and intra-processor communication. This makes it possible for applications to be more versatile and portable. In response to the needs of such applications, a virtual network device based on RPMsg has been added to the Reference Software Stack.

On the Safety Island side, a network device is created over an RPMsg endpoint with a specific service name. The RPMsg endpoint sends a Name Service message to the Primary Compute to announce its existence. The message is then processed by the RPMsg bus, which creates an RPMsg endpoint and a corresponding network device. Once this is done, the virtual network devices establish network communication.

On the Primary Compute side RPMsg frame must be copied to the Socket Buffer (skb) utilized by the Network Stack. However, if the traffic exceeds the performance limit, the Socket Buffer may get dropped during processing for congestion control or by the protocol layers. In such cases, the network statistics will increase the dropped packet counter.

In the above diagram, each Safety Island cluster has its own Shared Memory and MHUv3 device to communicate with the Primary Compute. The size of the Shared Memory is 16MB, and Safety Island Clusters 0, 1, and 2 have access to it. The Shared Memory instance has a Resource table (4KB), Vring 0, 1 (1MB each), and an RPMsg vbuffer (3MB) used to send and receive information between the Primary Compute and the Safety Island cluster.

On the Primary Compute, the Safety Island Remoteproc driver and RPMsg-based virtual interface driver are added to communicate with the Safety Island. The RPMsg-net driver on the Primary Compute and Veth-RPMsg on the Safety Island clusters implement the virtual ethernet device that is the basis for communication between the Primary Compute and Safety Island clusters.

Safety Island Remoteproc Driver

The Remoteproc framework allows different platforms/architectures to control (power on/off, load firmware) remote processors while abstracting the hardware differences, so the entire driver doesn't need to be duplicated. The Remoteproc platform driver is added to the RD-Kronos Stack to provide support for communication between Primary Compute and Safety Island clusters.

In the Kronos FVP, Linux running in the Primary Compute, regards the Safety Island clusters as its remote processors. The Kronos FVP Safety Island has three clusters. Each cluster behaves as an independent entity and has its own resources to establish the connection to the Primary Compute.

These clusters cannot be booted by the Primary Compute processor because they need to monitor the other hardware, including the Primary Compute. Therefore, the initial status of the clusters in the driver is `RPROC_DETACHED`, which means the cluster has been booted independently from the Primary Compute processor. This driver implements the

notification handler using an MHUv3 based mailbox, which notifies other cores when new messages are sent to the virtual queue.

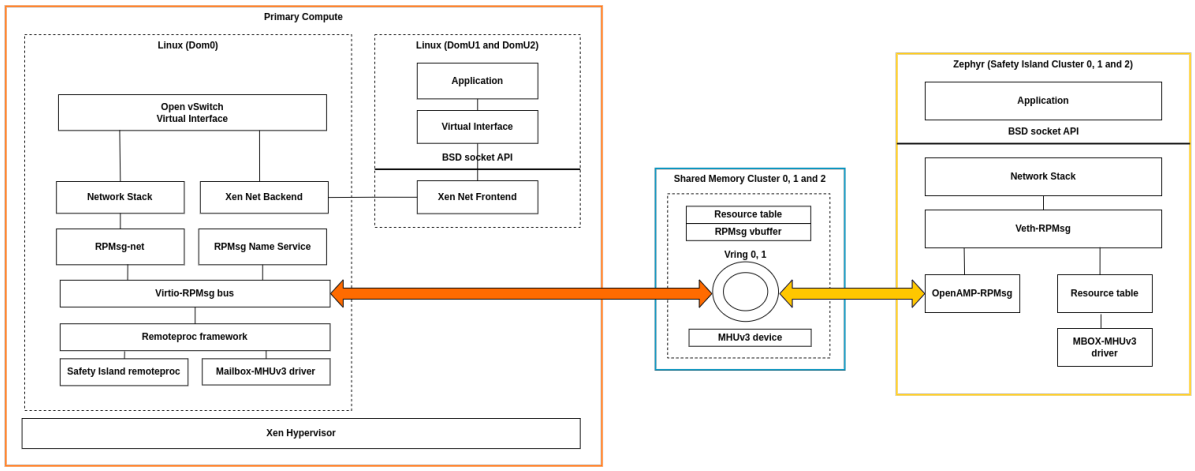
The Resource table, Vring 0, 1, and RPMsg vbuffer memory regions are set up in the device tree bindings for each cluster. The driver reads the device tree node for each cluster and adds it to the Remoteproc framework. Each cluster has its own Resource table, Vring 0, 1, and RPMsg vbuffer, which serve as the foundation for communication.

Virtualization Architecture

In the Virtualization Architecture of the Reference Software Stack, virtual network interfaces based on Xen drivers created in the control domain (Dom0) are exposed to the domUs. These virtual network interfaces are added to an Open vSwitch virtual switch along with an RPMsg Virtual Interface to communicate with the Safety Island.

Dom0 has a communication channel with the Safety Island which is the same as the Baremetal Architecture.

Arm Kronos Reference Software Stack HIPC - Virtualization Architecture



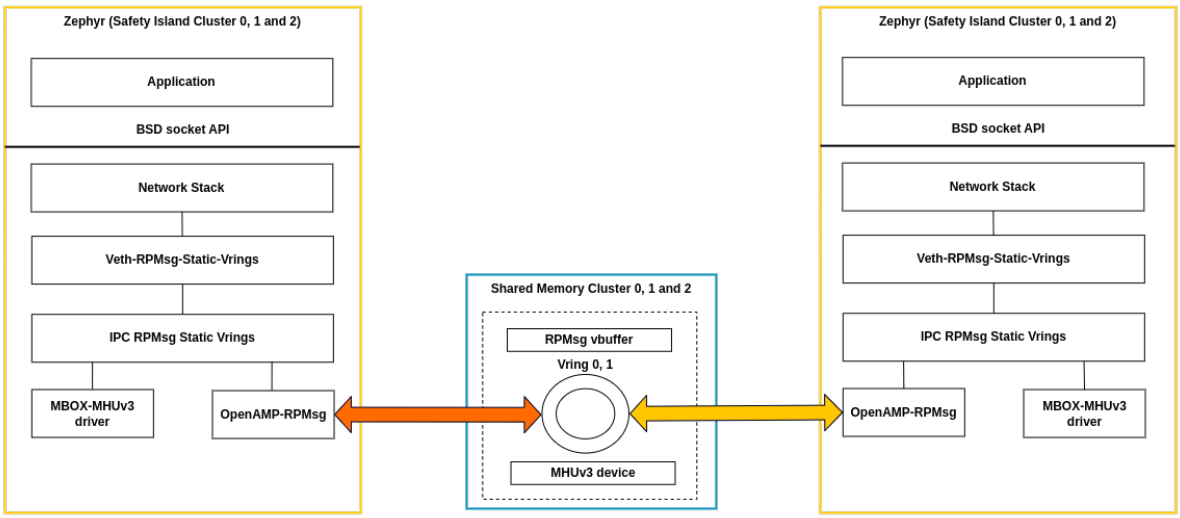
There are some limitations of the virtual network device over RPMsg. Refer to the release notes *Limitations* section.

3.5.3 Communication between the Safety Island clusters

Virtual Network Device over IPC Static Vrings

Zephyr IPC Service based virtual network devices are added to each cluster to provide communication between clusters via BSD sockets. The backend used for the IPC service is RPMSg Static Vrings. The IPC RPMsg Static Vrings backend is implemented on top of Virtio based RPMsg communication.

Inter-Safety Island Clusters Communication



3.5.4 Memory Map

The dedicated SRAM used by the Primary Compute and Safety Island Clusters 0, 1, and 2 for inter-processor data transfer has the following memory regions: **Resource table**, **Vring0**, **Vring1**, and **Virtio Buffer**.

Safety Island side:

Cluster 0:

Primary Compute <-> Cluster 0:

- `rsc_table` : Used to share resource information between Primary Compute and Cluster 0
- `shared_data` : Used for data transfer between Primary Compute and Cluster 0

Cluster 0 <-> Cluster 1, 2:

- `local_sram_cl0_cl1` : Used for data transfer between Cluster 0 and Cluster 1
- `local_sram_cl0_cl2` : Used for data transfer between Cluster 0 and Cluster 2

Refer to the device tree overlay below for more information about the memory addresses and region sizes.

- components/safety_island/zephyr/src/overlays/hipc/fvp_rd_kronos_safety_island_c0.overlay.

Cluster 1:

Primary Compute <-> Cluster 1:

- `rsc_table` : Used to share resource information between Primary Compute and Cluster 1

- `shared_data` : Used for data transfer between Primary Compute and Cluster 1

Cluster 1 <-> Cluster 0, 1:

- `local_sram_cl1_cl0` : Used for data transfer between Cluster 1 and Cluster 0
- `local_sram_cl1_cl2` : Used for data transfer between Cluster 1 and Cluster 2

Refer to the device tree overlay below for more information about the memory addresses and region sizes.

- components/safety_island/zephyr/src/overlays/hipc/fvp_rd_kronos_safety_island_c1.overlay.

Cluster 2:

Primary Compute <-> Cluster 2:

- `rsc_table` : Used to share resource information between Primary Compute and Cluster 2
- `shared_data` : Used for data transfer between Primary Compute and Cluster 2

Cluster 2 <-> Cluster 0, 2:

- `local_sram_cl2_cl0` : Used for data transfer between Cluster 2 and Cluster 0
- `local_sram_cl2_cl1` : Used for data transfer between Cluster 2 and Cluster 1

Refer to the device tree overlay below for more information about the memory addresses and region sizes.

- components/safety_island/zephyr/src/overlays/hipc/fvp_rd_kronos_safety_island_c2.overlay.

Primary Compute side:

- `si_c0_rproc_rsctbl` : Used to share resource information between Primary Compute and Cluster 0
- `si_c0_vdev0vring0` : Primary Compute vring, used to pass messages from Cluster 0 to Primary Compute
- `si_c0_vdev0vring1` : Safety Island Cluster 0 vring, used to pass messages from Primary Compute to Cluster 0
- `si_c0_vdev0buffer` : Used for data transfer between Primary Compute and Cluster 0
- `si_c1_rproc_rsctbl` : Used to share resource information between Primary Compute and Cluster 1
- `si_c1_vdev0vring0` : Primary Compute vring, used to pass messages from Cluster 1 to Primary Compute
- `si_c1_vdev0vring1` : Safety Island Cluster 1 vring, used to pass messages from Primary Compute to Cluster 1
- `si_c1_vdev0buffer` : Used for data transfer between Primary Compute and Cluster 1
- `si_c2_rproc_rsctbl` : Used to share resource information between Primary Compute and Cluster 2
- `si_c2_vdev0vring0` : Primary Compute vring, used to pass messages from Cluster 2 to Primary Compute
- `si_c2_vdev0vring1` : Safety Island Cluster 2 vring, used to pass messages from Primary Compute to Cluster 2

- `si_c2_vdev0buffer` : Used for data transfer between Primary Compute and Cluster 2

Refer to the device tree below for more information about the memory address and region size.

- meta-arm-bsp/recipes-bsp/trusted-firmware-a/files/fvp-rd-kronos/rdkronos.dts.

3.5.5 Network Topology

VLAN

Open vSwitch is used to create a virtual switch that connects all the network interfaces of the Primary Compute.

VLAN is a concept standardized by IEEE 802.1Q. It is used to partition a switch into multiple logical switches. The VLAN tag has a value from 0 to 4096 stored in the packet header. Usually 0 means that the packet is untagged, but some values are reserved.

On a switch, using VLAN tagged traffic makes sure that a packet tagged with a certain VLAN identifier reaches only ports that are configured to manage the traffic tagged with that identifier (tag).

The traffic between the Primary Compute and the Safety Island is using the following VLAN identifiers:

- VLAN **100**: Traffic from/to **Safety Island Cluster 0**
- VLAN **200**: Traffic from/to **Safety Island Cluster 1**
- VLAN **300**: Traffic from/to **Safety Island Cluster 2**

gPTP

Generalized Precision Time Protocol (gPTP) is a concept standardized by IEEE 802.1AS. It is used to synchronize the clocks of multiple systems over a network. A "PTP Instance" is an instance of this protocol. Each PTP Instance can have one or more logical access point to the network (a "PTP Port"). The source of the synchronized time in a domain is a single PTP Instance, the "Grandmaster PTP Instance", which always act as a server.

In the Kronos Reference Software Stack, Grandmaster PTP Instances are deployed on the Primary Compute (in Dom0 in case of the Virtualization Architecture), advertizing a single source of time to the other PTP Instances (on the Safety Island clusters and the DomUs) acting as clients. The Grandmaster PTP Instances each have one PTP Port per remote PTP Instance. All the Operating Systems that make use of gPTP have a dedicated service to handle the network messages:

- On Linux, the Linux PTP Project provides a `ptp4l` program that creates a PTP Port on a specified network interface. At system boot, one `ptp4l` daemon is started per network interface specified in the `LINUXPTP_IFACES` bitbake variable. This variable is set per *Use-Case*, with the Safety Island Communication Demo Use-Case making use of gPTP on all Operating Systems. The network interfaces created by Open vSwitch are not capable of software timestamping; hence, the direct network interfaces to the remote participant are used instead (for example for Safety Island Cluster 0, `ptp4l` binds to ethsi0, not brsi0). Note that `ptp4l` only writes to the system logger, not to the console, including in case of de-synchronization.
- On Zephyr, the kernel provides a Zephyr gPTP subsystem. Enabling it is done per application, by including the appropriate configuration file from components/safety_island/zephyr/src/overlays/gptp. They disable the Grandmaster capability and create a single PTP Port, on the first network interface. When the client is not synchronized with the server, the gPTP subsystem prints a warning-level logging message (`<wrn> net_gptp: Reset Pdelay requests`) at each tick of its state machine (about once per second).

In the Kronos Reference Software Stack, all of the PTP Instances use software timestamping. This limits the maximum achievable precision of the clock synchronization and it makes the stability of the clock vulnerable to software activity on either side of the gPTP link.

See *Integration Tests Validating gPTP* for details on how the functionality is validated.

External Connection

The Safety Island has a single network interface leading outside the Kronos FVP system located on Cluster 0.

A software-based network bridge deployed on Cluster 0 bridges this external interface with the IPC channels to the other Safety Island clusters so Cluster 1 and 2 can reach outside Kronos FVP.

See *Safety Island Cluster 0 Bridge* for more information.

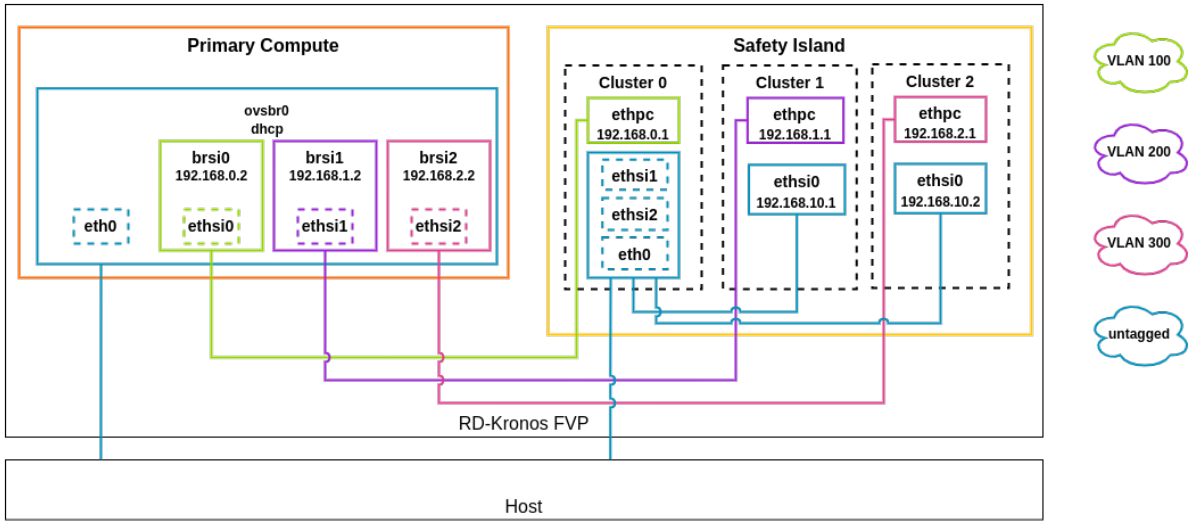
Baremetal Architecture

This diagram shows the network topology for the Baremetal Architecture. ethsi{N} is the name of the RPMsg-based Virtual Interfaces that are connected to Safety Island Cluster{N}, where N is the cluster number. For example, the ethsi0 interfaces are connected to Safety Island Cluster 0. Similarly, ethpc is the name of the interfaces that are connected to the Primary Compute.

ovsbr0 is the Open vSwitch network switch which carries untagged traffic. The communication between the Primary Compute and Safety Island is managed through the brsi{N} VLAN tagged switches that are configured to carry VLAN tagged traffic from/to the ethsi{N} interface with the Safety Island.

User space applications on the Primary Compute can communicate with Safety Island Cluster N via brsi{N}.

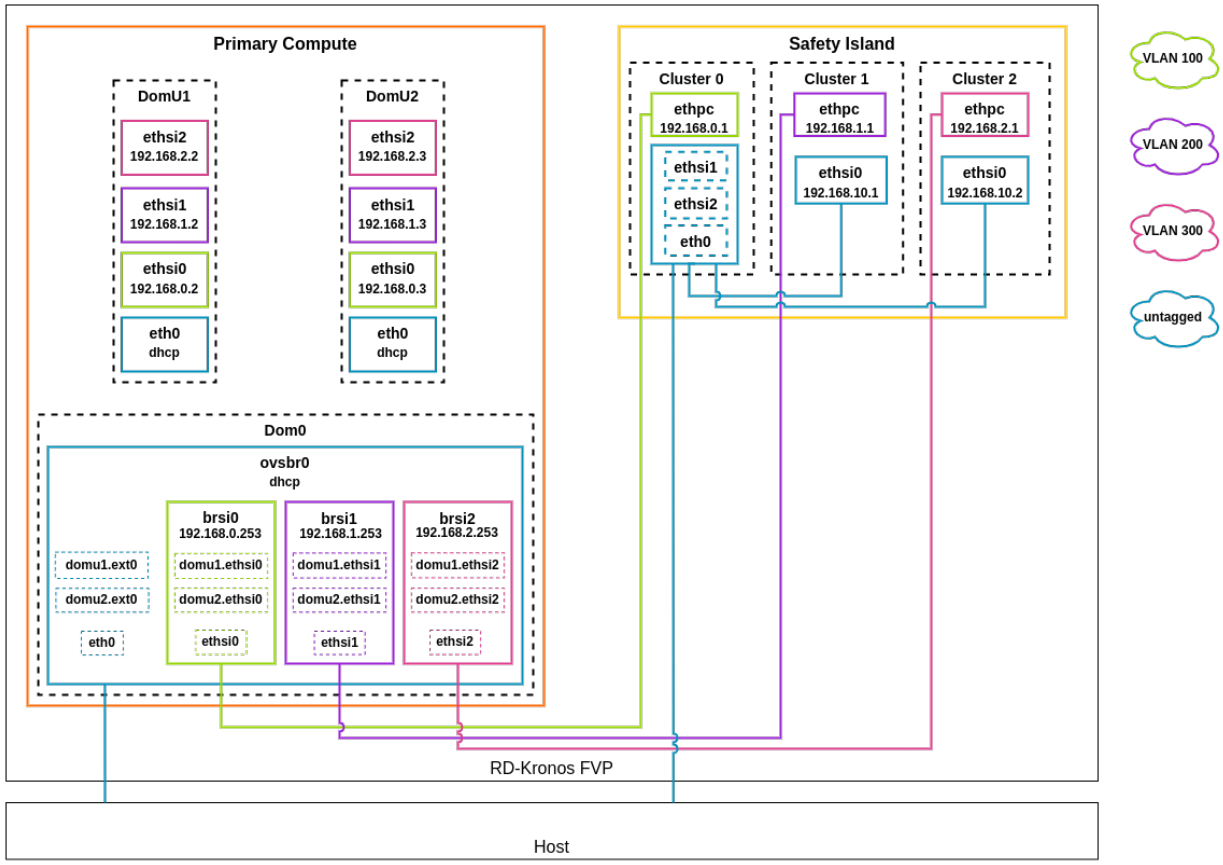
Arm Kronos Reference Software Stack Network Topology - Baremetal Architecture



Virtualization Architecture

As shown in the diagram below the virtual network interfaces for the Xen guests are based on Xen drivers. domu1.ethsi{N} and domu2.ethsi{N} are backend virtual network interfaces that are exposed to DomU1 and DomU2 guests. ethsi{N} in the Primary Compute is the RPMsg-based Virtual Interface that is connected to Safety Island Cluster{N} to provide communication between Primary Compute and Safety Island. ethsi{N}(Primary Compute) and domu1.ethsi{N} are added to Open vSwitch (brsi{N}) to have a connection between Dom0, DomU1 and Safety Island Cluster N.

Arm Kronos Reference Software Stack Network Topology - Virtualization Architecture



3.5.6 Device Tree

In Linux, a Remoteproc binding is needed for Safety Island clusters. It includes MHUv3 transmit/receive channels for signaling and several memory regions for data exchange. Each Safety Island cluster has it own Remoteproc binding that includes MHUv3 and Shared Memory.

The Linux device tree with the appropriate nodes for HIPC is located at meta-arm-bsp/recipes-bsp/trusted-firmware-a/files/fvp-rd-kronos/rdkronos.dts.

In Zephyr, there is an overlay device tree for the network over RPMsg application, which also defines the MHUv3 channels and device memory regions.

The Zephyr overlay device tree for FVP the Kronos board is located at components/safety_island/zephyr/src/overlays/hipc.

3.6 Components

The Reference Software Stack comprises of the following main components:

Component	Version	Source
RSS (Trusted Firmware-M)	53aa78efef274b9e46e63b429078ae1863609728 (based on master branch post v1.8.1)	Trusted Firmware-M repository
SCP-firmware	cc4c9e017348d92054f74026ee1beb081403c168 (based on master branch post v2.13.0)	SCP-firmware repository
Trusted Firmware-A	2.8.0	Trusted Firmware-A repository
OP-TEE	3.22.0	OP-TEE repository
Trusted Services	08b3d39471f4914186bd23793dc920e83b0e3197 (based on main branch, pre v1.0.0)	Trusted Services repository
U-Boot	2023.07.02	U-Boot repository
Xen	4.18	Xen repository
Linux Kernel	6.1.73	Linux repository and Linux preempt-rt repository
Zephyr	3.5.0	Zephyr repository

3.6.1 RSS

The Runtime Security Engine (RSE) is a security subsystem, which additionally adds an isolated environment to provide platform security services.

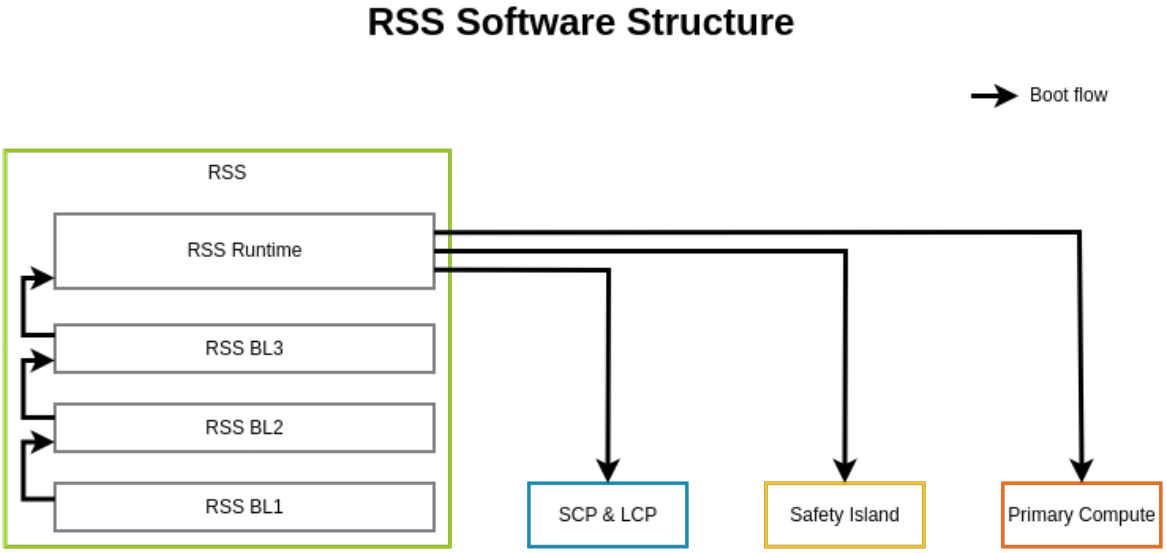
The RSS serves as the Root of Trust for the system, offering critical platform security services and holding and protecting the most sensitive assets in the system.

In the current software stack, the RSS offers:

- Secure boot, further details of which can be found in the TF-M Secure boot documentation.
- Crypto Service, which provides an implementation of the PSA Crypto API in a PSA Root of Trust (RoT) secure partition, further details of which can be found in the TF-M Crypto Service documentation.
- Internal Trusted Storage (ITS) Service, which is a PSA RoT Service for storing the most security-critical device data in internal storage that is trusted to provide data confidentiality and authenticity. Further details can be found in the TF-M Internal Trusted Storage Service documentation.

- Protected Storage (PS) Service, which is an Application RoT service that allows larger data sets to be stored securely in external flash, with the option for encryption, authentication and rollback protection to protect the data-at-rest. It provides an implementation of the PSA Secure Storage API in a PSA RoT secure partition. Further details can be found in the TF-M Internal Trusted Storage Service documentation.

The RSS internally consists of 3 boot loaders and a runtime. The following diagram illustrates the high-level software structure of the RSS and some relevant external components.



The *Secure Services* section provides more details of the RSS Runtime and the relevant components.

Memory Map

The Runtime Security Subsystem (RSS) maps the Primary Compute, System Control Processor (SCP), and Safety Island Clusters 0, 1, and 2 system memory regions via an Address Translation Unit (ATU) device to dedicated address spaces. This mapping allows access to those components memories and enables the transfer of the boot images.

From	To	Region
0x0 0040 0000 0000	0x0 FFFF FFFF FFFF	Primary Compute Address Space
0x1 0000 0000 0000	0x1 0000 FFFF FFFF	System Control Processor Address Space
0x2 0001 2000 0000	0x2 0001 3FFF FFFF	Safety Island Cluster 0 Address Space
0x2 0001 4000 0000	0x2 0001 5FFF FFFF	Safety Island Cluster 1 Address Space
0x2 0001 6000 0000	0x2 0001 7FFF FFFF	Safety Island Cluster 2 Address Space

Boot Loaders

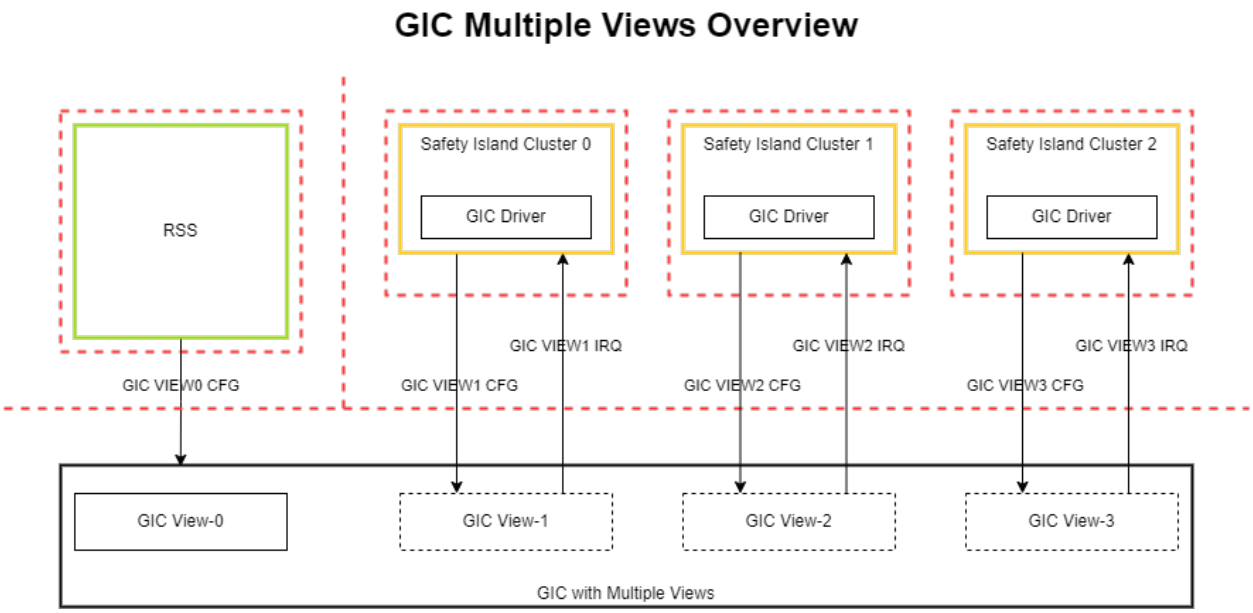
Refer to *RSS-oriented Boot Flow* for more details on the boot process.

Runtime

The RSS Runtime provides Crypto Service, PS Service and ITS Service as described above. See *Secure Services* for more details.

GIC Multiple Views

The GIC has a new optional feature which is intended to be used in mixed criticality systems. This feature provides multiple programming views which can be used by multiple operating systems.



For the RD-Kronos platform, Safety Island GIC provides 4 programming views:

- View-0: Used by RSS to configure View-1/2/3 for Safety Island Cluster-0/1/2.
- View-1: Used by Operating System on Safety Island Cluster-0.
- View-2: Used by Operating System on Safety Island Cluster-1.
- View-3: Used by Operating System on Safety Island Cluster-2.

Arm® CoreLink™ NI-710AE Network-on-Chip Interconnect

The CoreLink NI-710AE Network-on-Chip Interconnect is a highly configurable AMBA®-compliant system-level interconnect that enables functional safety for automotive and industrial applications. On the RD-Kronos platform, the NI-710AE handles traffic from four managers, i.e. Safety Island CPU cluster 0/1/2 and the RSS. It provides capabilities for these managers to access their corresponding subordinates. It also provides the capabilities for the subordinates to be exclusive to a certain manager or be shared among multiple managers during the different stages of RSS booting.

On Kronos, the configuration of NI-710AE is split to two stages, namely the discovery stage and the programming stage, both stages are done in RSS BL2. In the discovery stage, software can determine the structure of the NI-710AE domains, components, and subfeatures without previous knowledge of the configuration, based on the the base address of the configuration space. Then, the pre-defined APU tables are programmed to the APUs of the NI-710AE interfaces, and the RSS BL2 continues its normal boot process.

Downstream Changes

Patches for the RSS are included at meta-arm-bsp/recipes-bsp/trusted-firmware-m/files/fvp-rd-kronos/ to:

- Implement the RD-Kronos platform port, based on RD-Fremont.
- Load and boot the SCP.
- Load and boot the Safety Island.
- Load and boot the LCP.
- Load and boot the AP.
- Configure GIC View-1/2/3 for Safety Island.
- Configure the NI-710AE of the Safety Island.
- Support the runtime services listed above.
- Add Secure Firmware Update support for RSS, SCP, LCP, Safety Island and Primary Compute.
- Add a shutdown handler to be able to shutdown the FVP.

3.6.2 SCP-firmware

The Power Control System Architecture (PCSA) describes how systems can be built to provide microcontrollers to abstract various power, or other system management tasks, away from Primary Compute (PC).

The System Control Processor (SCP) Firmware provides a software reference implementation for the System Control Processor (SCP) and Local Control Processor (LCP) components.

System Control Processor (SCP)

For the RD-Kronos platform, the SCP software is deployed on a Cortex-M7 CPU.

The functionality of the SCP includes:

- Initialization of the system to manage Primary Compute (PC) boot
- **Runtime services:**
  - Power domain management
  - System power management
  - Performance domain management (Dynamic Voltage and Frequency Scaling)
  - Clock management
  - Sensor management
  - Reset domain management
  - Voltage domain management
- System Control and Management Interface (SCMI, platform-side)

Local Control Processor (LCP)

For the RD-Kronos platform, the Local Control Processor (LCP) software is deployed on Cortex-M55 CPUs.

The LCP is introduced for each application core to support a scalable power control solution in systems with very high core counts by SCP management. Now, the main functionality of the LCP is limited Per-core Dynamic Voltage Frequency Scaling (DVFS).

To minimize potential fault sources in a subsystem which functions in a mostly full-on state for the targeted application, the per core voltage scaling of DVFS is not supported.

The per core frequency scaling is supported with limitation. Only one Phase-Locked Loop (PLL) function (which may incorporate redundancy as a safety mechanism) is supported for the application processors. This limitation also minimizes potential fault sources.

MHUv3 Communication

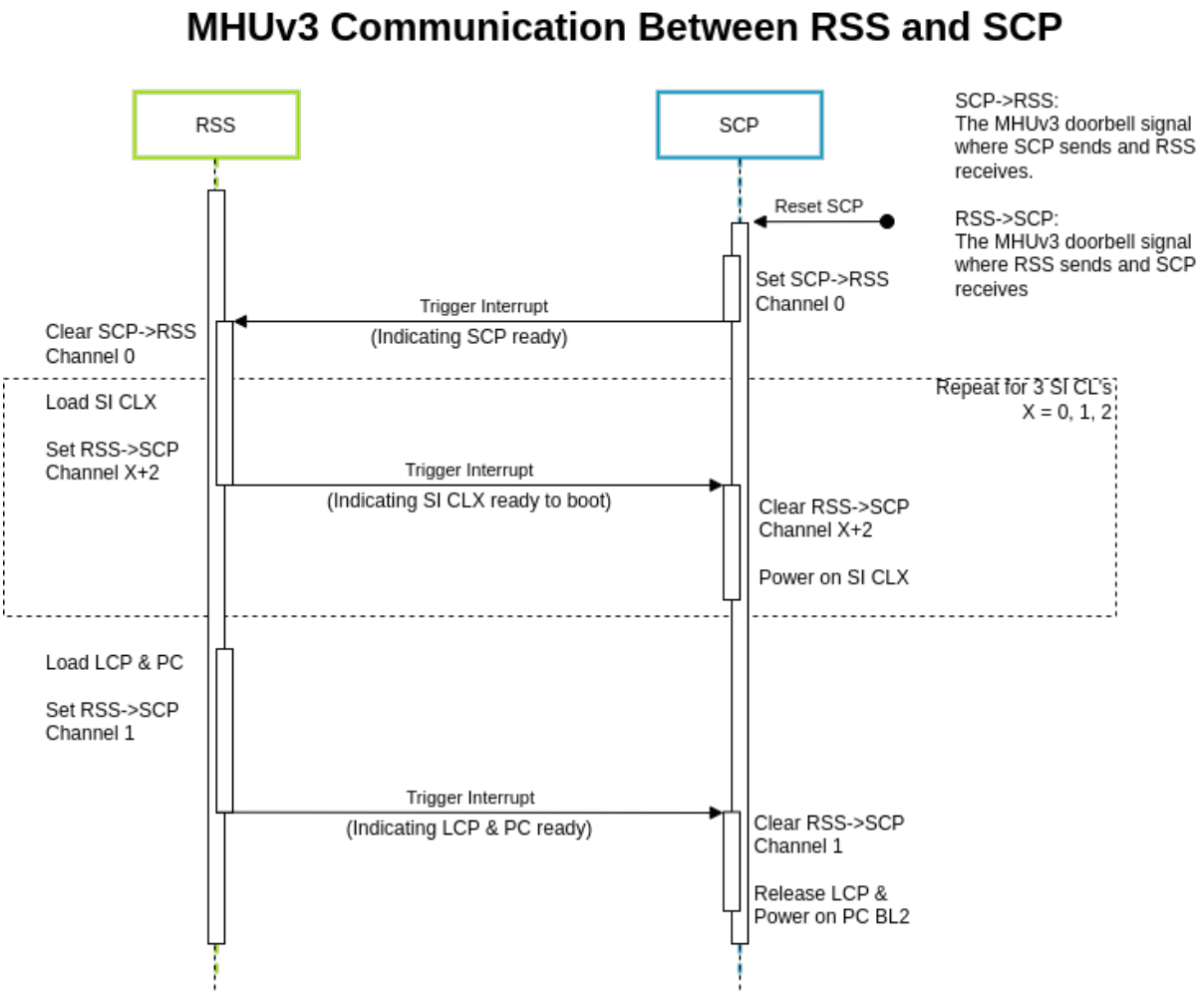
There are MHUv3 devices between the Cortex-M core where the RSS runs and the Cortex-M core where SCP-firmware runs. In the transport layer of MHUv3, doorbell signals are exchanged between the RSS and SCP.

For RD-Fremont platform, MHUv3 signals are sent:

- From SCP to the RSS to indicate that SCP has booted successfully
- From the RSS to SCP to indicate the LCP and Primary Compute (PC) is ready to boot

For RD-Kronos platform, the MHUv3 communication is extended for booting Safety Island (SI) clusters. The RSS sends a doorbell signal to SCP to notify that the image of a Safety Island cluster has been loaded to LLRAM and the cluster is ready to boot.

The following diagram illustrates the MHUv3 communication sequence between the RSS and SCP.



Downstream Changes

Patches for the SCP are included at meta-arm-bsp/recipes-bsp/scp-firmware/files/fvp-rd-kronos/ to:

- Implement the RD-Kronos platform port, based on RD-Fremont.
- Communicate with RSS via MHUv3 to conduct the boot flow.
- Power on Safety Island.
- Reset LCP.
- Power on PC.
- Add Primary Compute and Safety Island shared SRAM to Interconnect memory region map.
- Add a shutdown handler to be able to shutdown the FVP.

3.6.3 Primary Compute

Device Tree

The RD-Kronos FVP device tree contains the hardware description for the Primary Compute. The CPUs, memory and devices are statically configured in the device tree. It is compiled by the Trusted Firmware-A Yocto recipe, bundled in the Trusted Firmware-A flash image at rest and used to configure U-Boot, Linux and Xen at runtime. It is located at meta-arm-bsp/recipes-bsp/trusted-firmware-a/files/fvp-rd-kronos/rdkronos.dts.

Trusted Firmware-A

Trusted Firmware-A (TF-A) is the initial bootloader on the Primary Compute.

For RD-Kronos, the initial TF-A boot stage is BL2, which runs from a known address at EL3, using the `BL2_AT_EL3` compilation option. This option has been extended for RD-Kronos to load the `FW_CONFIG` for dynamic configuration (a role typically performed by BL1). BL2 is responsible for loading the subsequent boot stages and their configuration files from the flash containing the FIP image, which contains:

- BL31
- BL32 (*OP-TEE*)
- BL33 (*U-Boot*)
- The `HW_CONFIG` device tree
- The `TB_FW_CONFIG` device tree
- The `TOS_FW_CONFIG` device tree

Downstream Changes

Patch files can be found at meta-arm-bsp/recipes-bsp/trusted-firmware-a/files/fvp-rd-kronos/ to:

- Implement the RD-Kronos platform port, based on RD-Fremont.
- Compile the HW_CONFIG device tree and add it to the FIP image.
- Extend BL2_AT_EL3 to load the FW_CONFIG for dynamic configuration.
- Support for the OP-TEE SPMC on the RD-Kronos platform.
- Add the following device tree nodes to the RD-Kronos platform.
  - PL180 MMC
  - PCIe controller
  - SMMUv3
  - HIPC
- Assign the shared buffer for the Management Mode (MM) communication between U-Boot and OP-TEE.
- Add Secure Firmware Update support for Primary Compute.

OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a Normal world Linux kernel running on Neoverse-V3AE cores using the TrustZone technology. OP-TEE implements TEE Internal Core API v1.1.x which is the API exposed to Trusted Applications and the TEE Client API v1.0, which is the API describing how to communicate with a TEE, further details of which can be found in the OP-TEE API Specification.

Downstream Changes

Patch files can be found at meta-arm-bsp/recipes-security/optee/files/optee-os/fvp-rd-kronos/ to:

- Implement the RD-Kronos platform port.
- Boot OP-TEE as SPMC running at SEL1.

Trusted Services

The Trusted Services project provides a framework for developing and deploying device root-of-trust services for A-profile devices. Alternative secure processing environments are supported to accommodate the diverse range of isolation technologies available to system integrators.

The Reference Software Stack implements the following Secure Services on top of the Trusted Services framework:

- Crypto Service
- Secure Storage Service
- UEFI SMM Services

See *Secure Services* for more information.

Downstream Changes

Patch files can be found at meta-arm-bsp/recipes-security/trusted-services/fvp-rd-kronos/ to:

- Implement the RD-Kronos platform port.
- Support MHUv3 doorbell communication.
- Support RSS communication protocol.
- Support crypto and secure storage backends for the RD-Kronos platform.
- Support transfer capsule update FF-A protocol.

U-Boot

U-Boot is the Normal world second-stage bootloader (BL33 in TF-A) on the Primary Compute. It consumes the `HW_CONFIG` device tree provided by Trusted Firmware-A and provides UEFI services to UEFI applications like Linux and Xen. The device tree is used to configure U-Boot at runtime, minimizing the need for platform-specific configuration.

In the current software stack, the U-Boot implementation of the UEFI subsystem uses the FF-A (Arm Firmware Framework for Arm A-profile) driver to communicate with the UEFI SMM Services in the Secure world to store and read UEFI variables that are stored in the Protected Storage Service provided by the RSS.

Downstream Changes

The implementation is based on the VExpress64 board family. Patch files can be found at meta-arm-bsp/recipes-bsp/u-boot/u-boot/fvp-rd-kronos/ to:

- Enable VIRTIO_MMIO and RTC_PL031 in the base model.
- Set max mmc block count to the limitation of PL180.
- Add MMC card to the BOOT_TARGET_DEVICES of FVP to support the scenarios of Linux/FreeBSD Distros installation.
- Move sev() and wfe() definitions to common Arm header file.
- Modify pending callback to test if transmit FIFO is empty in PL01x driver.
- Add support for SMCCCv1.2 x0-x17 registers.
- Introduce Arm FF-A support.
- Introduce armffa command.
- Add MM communication support using FF-A transport.
- Add Secure Firmware Update support.

Xen

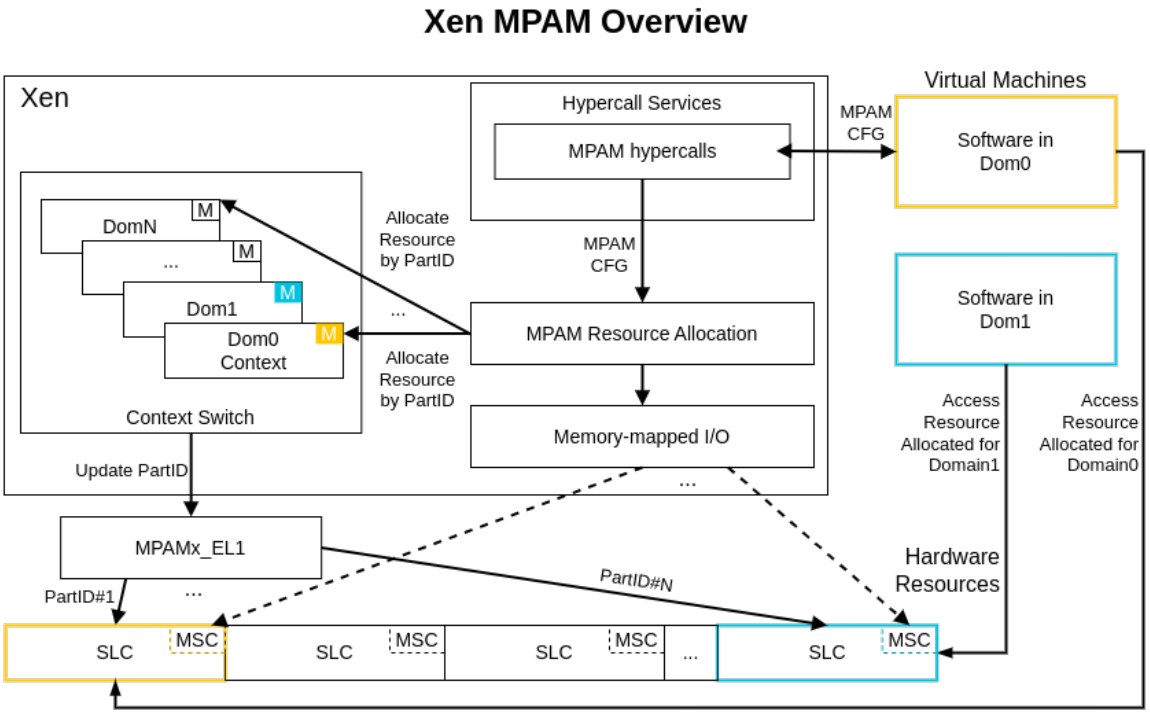
Xen is a type-1 hypervisor, providing services that allow multiple computer operating systems to execute on the same computer hardware concurrently. Responsibilities of the Xen hypervisor include memory management and CPU scheduling of all virtual machines (domains), and for launching the most privileged domain (Dom0) - the only virtual machine which by default has direct access to hardware. From the Dom0 the hypervisor can be managed and unprivileged domains (DomU) can be launched. Xen is only included in the Virtualization Reference Software Stack Architecture.

Boot Flow

On starting up, the GRUB2 configuration uses the "chainloader" command to instruct the UEFI services provider (U-boot) to load and run Xen as an EFI application. Further, Xen reads its configuration (`xen.cfg`) from the boot partition of the virtio disk containing the boot arguments for Xen and Dom0 to start the whole system.

MPAM

The Arm Memory Partitioning and Monitoring (MPAM) extension is enabled in Xen. MPAM is an optional extension to Arm® 8.4-A and later versions. It defines a method that software can utilize to apportion and monitor the performance-giving resources (usually cache and memory bandwidth) of the memory system. Domains can be assigned with dedicated system level cache (SLC) slices so that cache contention with multiple domains can be mitigated.



The stack offers several methods for users to configure MPAM for domains:

- For Dom0, an optional Xen command line parameter `dom0_mpam` can be used to configure the cache portion bit mask (CPBM) for Dom0. The format of the `dom0_mpam` parameter is:

```
dom0_mpam=slc:<CPBM in hexadecimal>
```

  To use the `dom0_mpam` parameter, users can add this parameter to the `options` of the `[xen]` section in xen.cfg config file. An example to assign the first 4 portions of SLC to Dom0 at Xen boot time is shown below:

```
[xen]
options=(...) dom0_mpam=slc:0xf
```

- Users can also apply MPAM configuration for guests at guest creation time by guest VM configuration file using an optional configuration `mpam`. An example is shown below:

```
mpam = ['slc=0xf']
```

- There is a set of sub-commands in "xl" to allow users to use MPAM at runtime. Users can use the `xl psr-hwinfo` command to query the system information of MPAM, and use `xl psr-cat-set` or `xl psr-cat-show` to configure or read the CPBM for Dom0 and DomU at runtime.

  The format of `xl psr-cat-set` is (`-l 0` refers to SLC):

```
xl psr-cat-set -l 0 <Domain ID> <CPBM in hexadecimal>
```

  The format of `xl psr-cat-show` is (`-l 0` refers to SLC):

```
xl psr-cat-show -l 0
```

More detailed information of the sub-commands, refer to the `--help` of each sub-command respectively.

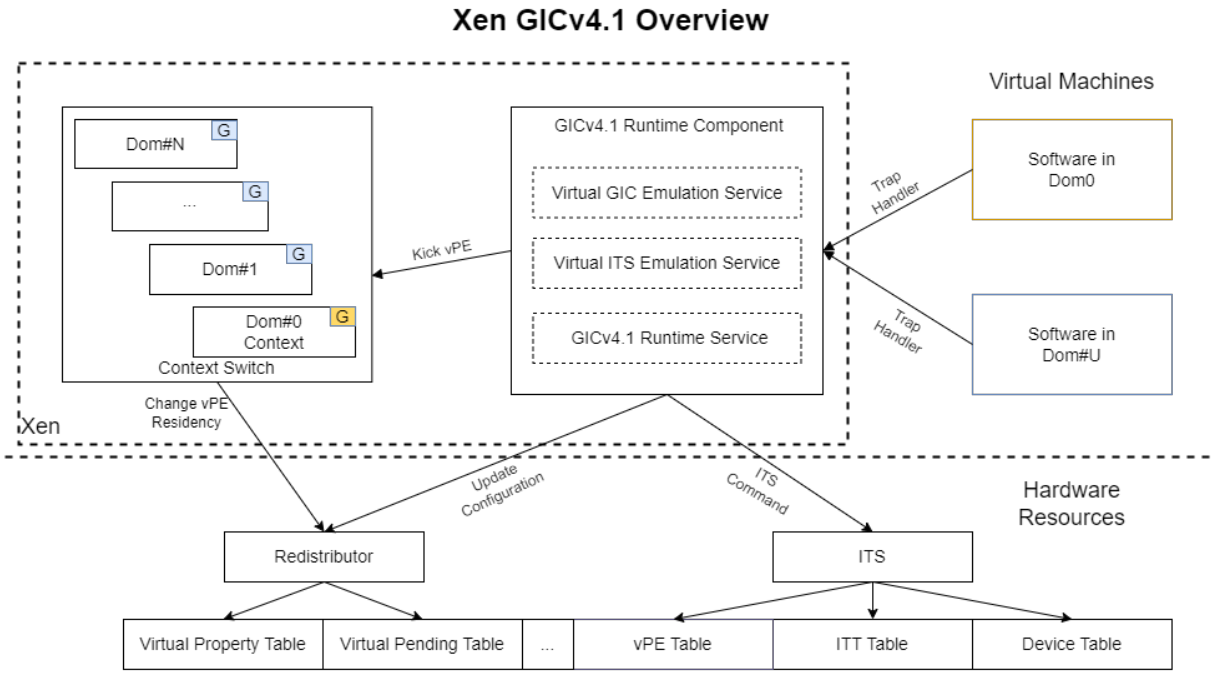
Limitations of MPAM support in Xen include:

- Currently, MPAM support in Xen is available for the system level cache (SLC) partitioning only.
- DomU MPAM settings can only be manipulated by xl after the DomU has been created and started.
- The FVP only provides the programmer's view of MPAM. There is no functional behaviour change implemented.

GICv4.1

The GICv4.1 - Direct injection of virtual interrupts (GICv4.1) is enabled in Xen. GICv4.1 is an extension to GICv3 with extra direct Virtual Locality-specific Peripheral Interrupt (vLPI) and Virtual Software-generated Interrupt (vSGI) injection enabled. This feature allows users to describe to the Interrupt Translation Service (ITS) how physical events map to virtual interrupts in advance. If the Virtual Processing Element (vPE) targeted by a virtual interrupt is running, the virtual interrupt can be forwarded without the need to first enter the Xen hypervisor. This can reduce the overhead associated with virtualized interrupts, by reducing the number of times the hypervisor is entered.

With Xen Kconfig `CONFIG_GICV4=y`, the Kronos platform will be automatically equipped with the capability of all GICv4.1 features.



The stack offers the PCI AHCI SATA Disk for users to utilize GICv4.1 vLPI direct injection for DomU1:

- Attach PCI AHCI SATA disk `ahci[0000:00:1f.0]` to DomU1 with static PCI passthrough method, by adding the following to the Dom0 Linux kernel command line:


```
xen-pciback.hide=(0000:00:1f.0)
```

In addition, the configuration for DomU1 shall also include a new line of `pci = ['0000:00:1f.0']` for enabling the PCI AHCI SATA disk.

For GICv4.1 vLPI/vSGI validation, refer to *GICv4.1 vLPI/vSGI Direct Injection Demo*.

SVE2

The Scalable Vector Extension version two (SVE2) is enabled in Xen. This feature is used as an extension to AArch64, to allow for flexible vector length implementations.

SVE vector length can be specified as an optional parameter along with enabling SVE2. The allowed values are from 128 to maximum 2048 limited by the hardware supported maximum SVE vector length. Dom0 and guest SVE settings follow the Arm Kronos Reference Design's maximum vector length of 128. These settings are set in yocto/meta-kronos/recipes-core/domu-package/domu-envs.inc and b/yocto/meta-kronos/recipes-extended/xen-cfg/xen-cfg.bb.

For more information on SVE2, refer to SVE2 guide. Xen command line options for SVE for dom0 can be found under xen-command-line options and SVE configuration for guests can be found under xl configuration.

For SVE2 validation, refer to *Integration Tests Validating SVE2*.

Downstream Changes

Patches for the Xen MPAM extension support, PCI Device Passthrough, and GICv4.1 Enablement at yocto/meta-kronos/recipes-extended/xen/files/ to:

- Discover MPAM CPU feature
- Initialize MPAM at Xen boot time
- Support MPAM in Xen tools to apply the domain MPAM configuration in userspace at runtime
- Support PCI Device Passthrough
- Discover GICv4.1 feature
- Initialize GICv4.1 at Xen boot time
- Support GICv4.1 features of vLPI and vSGI Direct Injection
- Support EFI capsule update from runtime and on disk

Linux Kernel

In the Baremetal Architecture, the Linux kernel is a real-time kernel that uses the PREEMPT_RT patch. In the Virtualization Architecture, both Dom0, DomU1 and DomU2 run a standard kernel.

Note: Here, the "standard kernel" is a terminology compared to a real-time kernel, a term borrowed from Kernel Types that are defined in the Yocto Project.

Remoteproc

In Linux, a remoteproc driver for the Safety Island is added to the Linux kernel. It is used to support RPMsg communication between the Arm® v9-A cores (from Primary Compute) and the Safety Island. More details on the communication can be found in the *HIPC* section.

Virtual Network over RPMsg

In order to allow applications to access the remote processor using network sockets, a virtual network device over RPMsg is introduced. The `rpmsg_net` kernel module is added for creating a virtual network device and converting RPMsg data to network data.

SVE2

The Scalable Vector Extension version two (SVE2) is enabled in Linux. This feature is used as an extension to AArch64, to allow for flexible vector length implementations.

For more information on SVE2, refer to SVE2 guide.

Downstream Changes

The `arm_si_rproc` and `rpmsg_net` drivers can be found at components/primary_compute/linux_drivers.

Additional patches are located at yocto/meta-kronos/recipes-kernel/linux/files related to:

- Making virtio rpmsg buffer size configurable
- Disable remoteproc virtio rpmsg to use DMA API in Xen guest
- Adding MHUv3 driver

3.6.4 Safety Island

Zephyr

Zephyr is an open source real-time operating system based on a small footprint kernel designed for use on resource-constrained and embedded systems.

The Reference Software Stack uses Zephyr 3.5.0 as a baseline and introduces a new board `fvp_rd_kronos_safety_island` for the Kronos FVP. It reuses the `fvp_aemv8r` SoC support and adds a pair of patches for MPU device region configuration.

The Zephyr image for this board is running on the Safety Island clusters. In order to enable communication with Armv9-A cores (from Primary Compute), a set of drivers are added into Zephyr by means of an out-of-tree module. More details on the communication can be found in the *HIPC* section.

MHUv3

The Arm Message Handling Unit Version 3 (MHUv3) is a mailbox controller for inter-processor communication. In the Kronos FVP, there are MHUv3 devices on-chip for signaling between Armv9-A and Safety Island clusters, using the doorbell protocol. A driver is added into the Zephyr mailbox framework to support this device.

Virtual Network over RPMsg

A `veth_rpmsg` driver is added for network socket based communication between Armv9-A and Safety Island clusters. It implements an RPMsg backend by the OpenAMP library and an adaptation layer for converting RPMsg data to network data.

Virtual Network over IPC RPMsg Static Vrings

A `ipc_rpmsg_veth` driver is added for network socket based communication between Safety Island clusters. It implements virtual network device based on IPC RPMsg Static Vrings.

Zperf sample

The zperf sample can be used to stress test inter-processor communication over a virtual network on the Kronos FVP. The board overlay dts and configuration file are added to this sample. This sample needs to be used together with iperf on the Armv9-A side for network performance testing.

Downstream Changes

The board support for `fvp_rd_kronos_safety_island` is located at components/safety_island/zephyr/src/boards/arm64/fvp_rd_kronos_safety_island.

The out-of-tree driver for virtual network over RPMsg is located at components/safety_island/zephyr/src/drivers/ethernet.

The out-of-tree driver for MHUv3 device is located at components/safety_island/zephyr/src/drivers/mbox.

Additional patches are located at yocto/meta-kronos/recipes-kernel/zephyr-kernel/files/zephyr related to:

- Configuring the MPU region
- Configuring and fixing VLAN
- Working around the shell interfering with network performance
- Adding zperf download bind capability
- Adding SMSC91x driver promiscuous mode
- Fixing connected datagram socket packet filtering
- Fixing race conditions in poll and condvar
- Fixing gPTP message generation correctness
- Fixing gPTP packet priority
- Conforming to the gPTP VLAN rules

3.7 Applications

3.7.1 Critical Application Monitoring Demo

Introduction

Critical applications often follow a pattern where the workloads are split into multiple periodic tasks chained together to produce a feature pipeline. Detection of application execution faults in such safety-critical systems is one of the pillars of a system's reliability strategy. The Critical Application Monitoring (CAM) project implements a solution for monitoring such critical applications using a monitoring service that runs on a higher safety level system. The main goal of CAM is to ensure that a certain piece of code running in critical applications executes periodically at a specific frequency. When the execution time is violated, critical applications are deemed as malfunctioning. The classes of issues that CAM can detect can be broadly classified into:

- Temporal issues: Events arriving outside the expected frequency.
- Logical issues: Events arriving out of order.

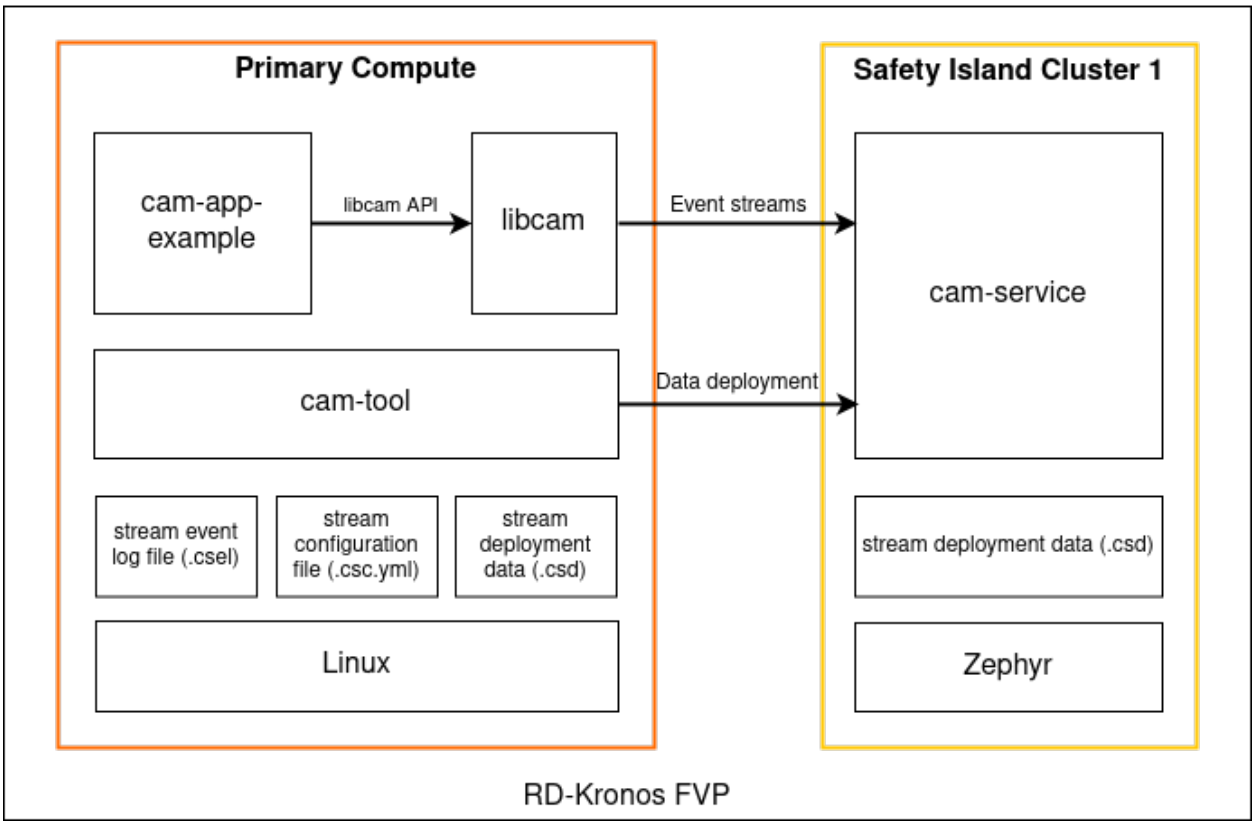
The CAM project is integrated into the Kronos Reference Software Stack to demonstrate the feasibility of monitoring Primary Compute applications from the Safety Island. Refer to Critical Application Monitoring Documentation for more information on CAM project and its implementation details.

Critical Application Monitoring on Kronos

The Critical Application Monitoring demo can be run on both Baremetal and Virtualization Architectures.

The following diagram shows the architecture of the demo in the Baremetal Architecture:

Critical Application Monitoring Demo High-Level Diagram



CAM consists of the following major components:

- **Stream configuration file**: Configuration file containing the number of stream events and their timing characteristics according to the requirements of the critical application.
- **Stream deployment data**: Binary representation of the stream configuration that needs to be deployed to the Safety Island.
- **cam-tool**: A python-based tool used to generate and deploy stream deployment data by analyzing stream configuration file.
- **cam-service**: CAM monitoring agent that monitors event streams sent by critical applications and runs from higher safety cores in the Safety Island. `cam-service` uses the stream deployment data to validate event streams produced by critical applications.
- **libcam**: CAM library that offers a simple, thread-safe API that can be used by critical applications to integrate the CAM project. The API enables the applications to register with `cam-service` and generate event streams to be sent to `cam-service`.
- **cam-app-example**: An example application that uses `libcam` API to integrate CAM framework. It also supports error injection into the stream events to trigger a fault detection by `cam-service`.

The Primary Compute components are deployed on the baremetal Linux root filesystem in the Baremetal Architecture build and on the DomU1 and DomU2 Linux root filesystem in the Virtualization Architecture.

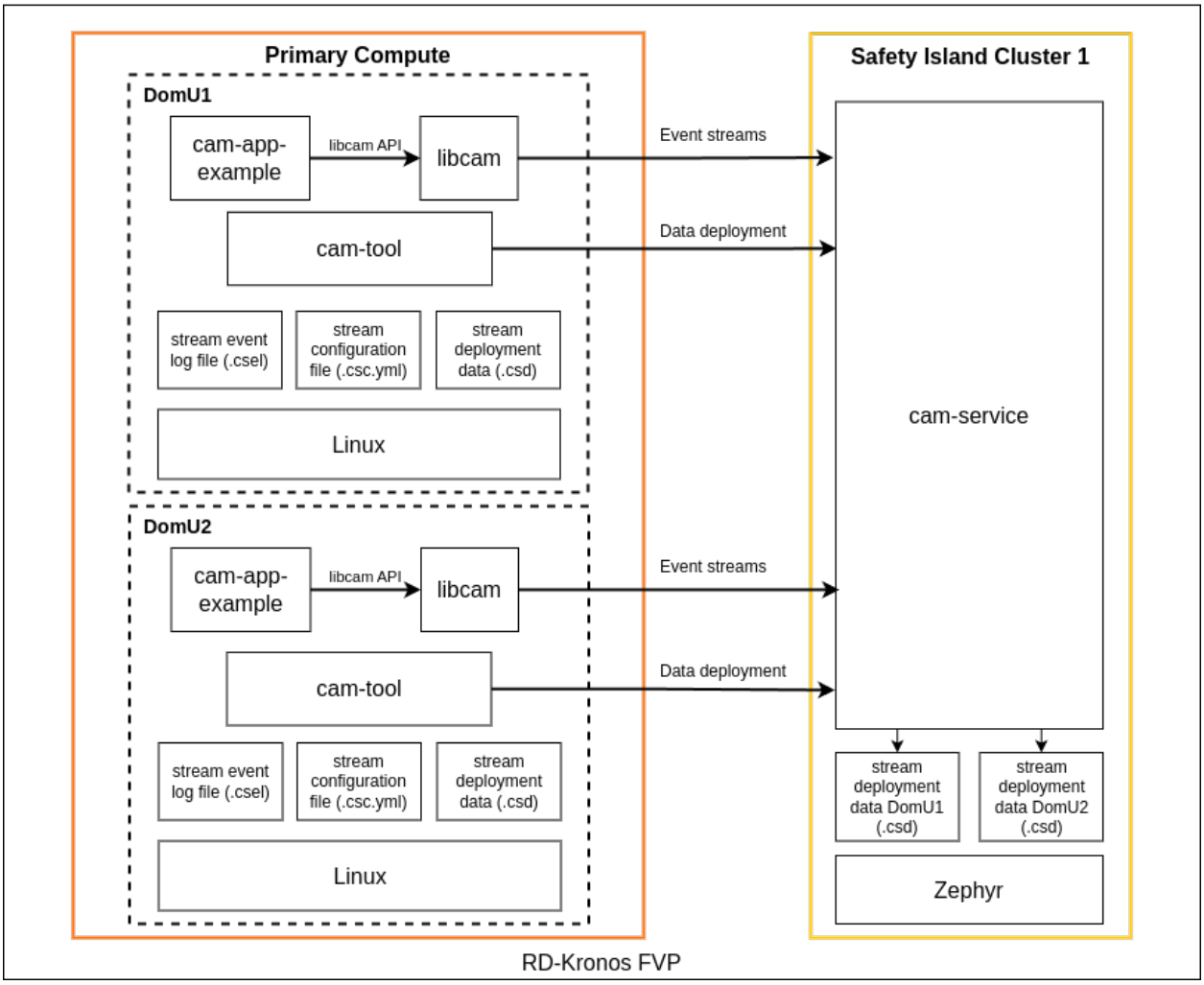
In the Kronos Reference Software Stack, `cam-service` is deployed on the Safety Island Cluster 1 in order to provide applications on the Primary Compute with a high safety level of monitoring services.

The following are platform requirements to support the `cam-service` deployment on the Safety Island:

- Communication between the Safety Island and the Primary Compute for event streams.
- Synchronized clocks on the Safety Island and the Primary Compute for temporal check.
- Storage and a file system on the Safety Island for stream data deployment.

Virtualization Architecture

The following diagram shows the architecture of the demo in the Virtualization Architecture:



In this deployment, two different instances of **cam-app-example** run on DomU1 and DomU2. Each application is monitored by **cam-service** concurrently via separate data deployment and event streams.

Communication Interfaces

BSD sockets (over TCP) are used in order to send the event message from `cam-app-example` to `cam-service` via the *Heterogeneous Inter-Processor Communication (HIPC)* feature.

Time Synchronization

Real-time clocks on the Primary Compute and the Safety Island are synchronized via the *gPTP* protocol.

Zephyr File System

Zephyr supports the FAT file system and can mount it to a RAM disk. Refer to Zephyr file system.

Note: Due to the volatility of the RAM disk, on every system boot, the CAM stream data needs to be deployed from the Primary Compute to the Safety Island Cluster 1 via `cam-tool`.

Validation

Refer to the CAM Demo validations *Integration Tests Validating the Critical Application Monitoring Demo*.

3.7.2 Safety Island Actuation Demo

Introduction

The Safety Island Actuation Demo is an example application that shows how an Autonomous Drive software stack can be run in a compute environment composed of a high-performance Primary Compute platform coupled with a higher reliability Safety Island.

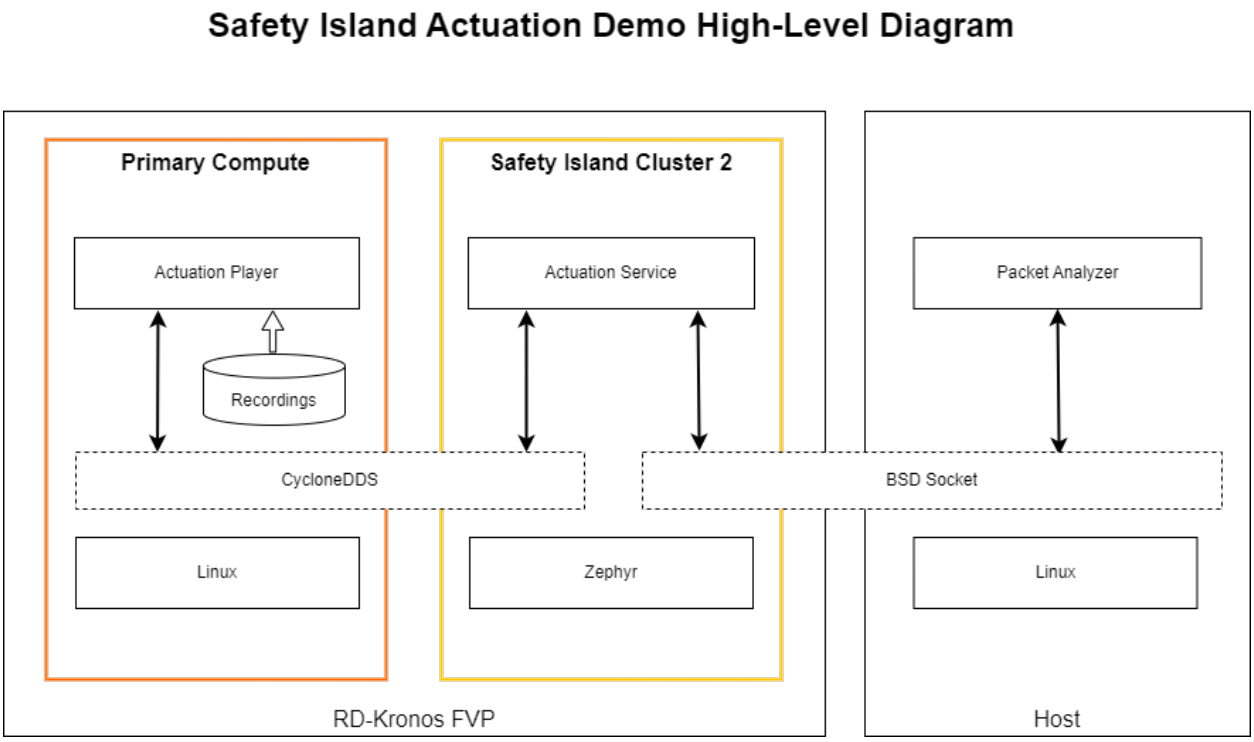
The Safety Island Actuation Demo features an "Actuation Service" application running on the Safety Island that receives inputs from the Primary Compute and generates control commands that can be passed to an actuation system. A reference implementation is provided by the Safety Island Actuation Demo. The software running on the Primary Compute is an Autoware pipeline and the "Actuation Service" takes the form of a Zephyr application showcasing the Pure Pursuit algorithm from Autoware.Auto. The two communicate via Data Distribution Service (DDS) messages over a network interface.

Safety Island Actuation Demo on Kronos

Compared to the default deployment of the Safety Island Actuation Demo, the Kronos deployment has, on the Primary Compute, an "Actuation Player" component instead of the Autoware pipeline and, on the Host, a "Packet Analyzer" instead of a visualization software. This is done in order to minimize the load for an FVP target.

This demo can run on both the Baremetal and Virtualization Architectures. In case of the Virtualization Architecture, the "Actuation Player" is deployed on DomU1.

The following diagram describes the data flow of the demo:



For more information on the underlying network topology, and the description of the network bridging involved in providing access to the external network, see the *Memory Map* section.

Main Components

The Actuation Demo on Kronos has 3 components:

- Actuation Player
  - Plays a recording of a driving scenario from the default deployment of the Actuation Demo
  - Recipe at yocto/meta-kronos/recipes-demos/actuation/actuation-player_2.0.0.bb
- Actuation Service
  - Functionally the same as the "Actuation Service" from the default deployment of the Actuation Demo, except that the computed commands are sent through a BSD socket instead of a DDS connection
  - Recipe at yocto/meta-kronos/recipes-kernel/zephyr-kernel/zephyr-actuation.bb
  - Zephyr overlays at:
    - components/safety_island/zephyr/src/overlays/hipc/fvp_rd_kronos_safety_island_c2.conf
    - components/safety_island/zephyr/src/overlays/hipc/fvp_rd_kronos_safety_island_c2.overlay
    - components/safety_island/zephyr/src/apps/actuation/boards/fvp_rd_kronos_safety_island_c2_actuation.conf
- Packet Analyzer
  - Checks for correctness of the "Actuation Service" output

- Recipe at yocto/meta-kronos/recipes-demos/actuation/packet-analyzer-native_2.0.0.bb

Communication Interfaces

Actuation Player <> Actuation Service

CycloneDDS (Using a specific upstream commit located at yocto/meta-kronos/recipes-demos/actuation/cyclonedds_0.10.3.inc) is used for the communication between the "Actuation Player" and the "Actuation Service".

Actuation Service <> Packet Analyzer

BSD socket (TCP Protocol) is used in order to send the Control Commands from the "Actuation Service" to the "Packet Analyzer".

Validation

Refer to the Actuation Demo validations *Integration Tests Validating the Safety Island Actuation Demo*

3.7.3 Safety Island Cluster 0 Bridge

Introduction

The Safety Island Cluster 0 Bridge is a software application running on the Safety Island Cluster 0 that acts as a network bridge connecting together the network interfaces with the other Safety Island clusters and the network interface with the Host. The code for the application can be found at components/safety_island/zephyr/src/apps/bridge.

Architecture

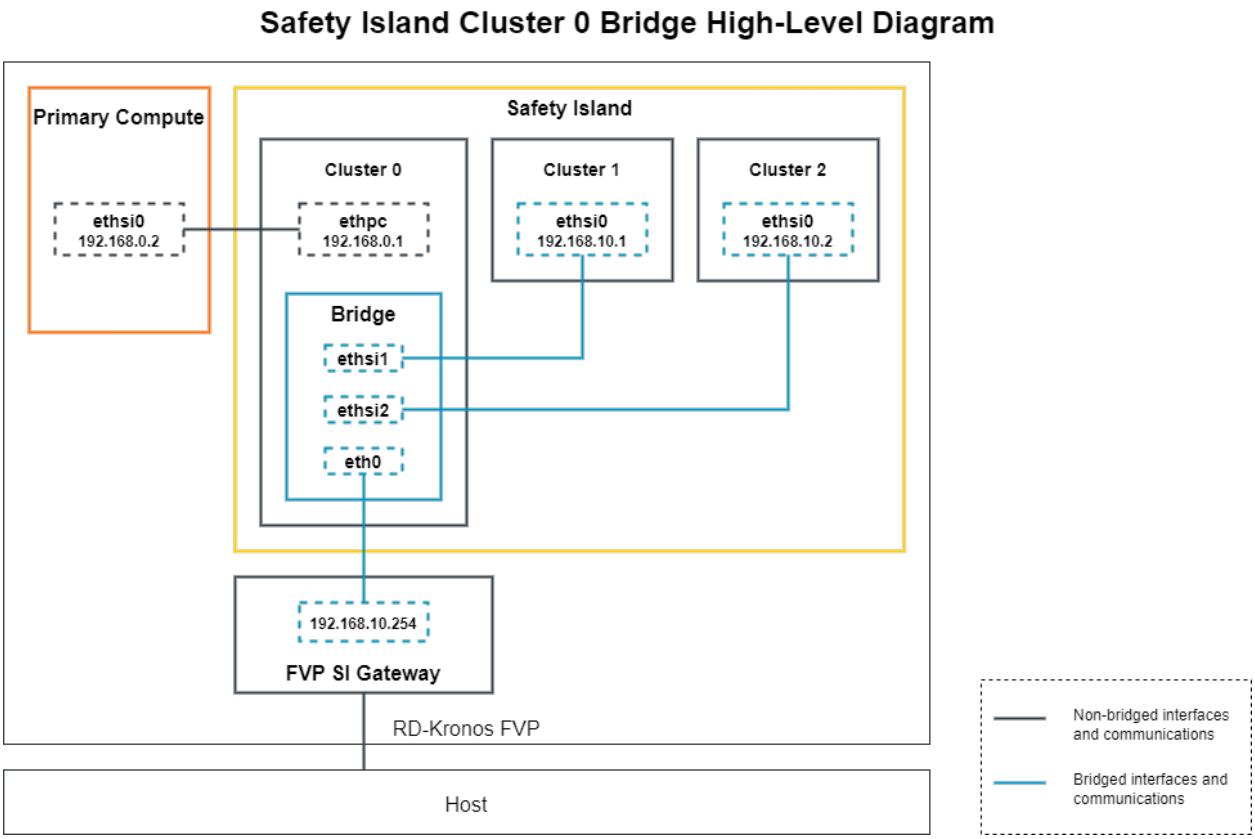
Components

The bridge is a Zephyr application making use of its Ethernet Bridging API.

It is not VLAN-aware, as none of the traffic on the bridged networks (the external network and all the inter-cluster networks) is VLAN-tagged.

The current Zephyr bridge functionality does not feature a learning process, which means that incoming packets on a registered interface are sent to all other registered interfaces at all times.

Diagram



Interfaces

The network interfaces added to the bridge forward all incoming packets to the other registered interfaces without processing the packets. It means that those interfaces can't respond to Internet Control Message Protocol (ICMP) *echo requests* from other network nodes and can't be used for other purposes by the application.

The network interface to the Primary Compute can respond to ICMP *echo requests* but has no functional use.

Validation

See *Integration Tests Validating the Safety Island Cluster 0 Bridge*.

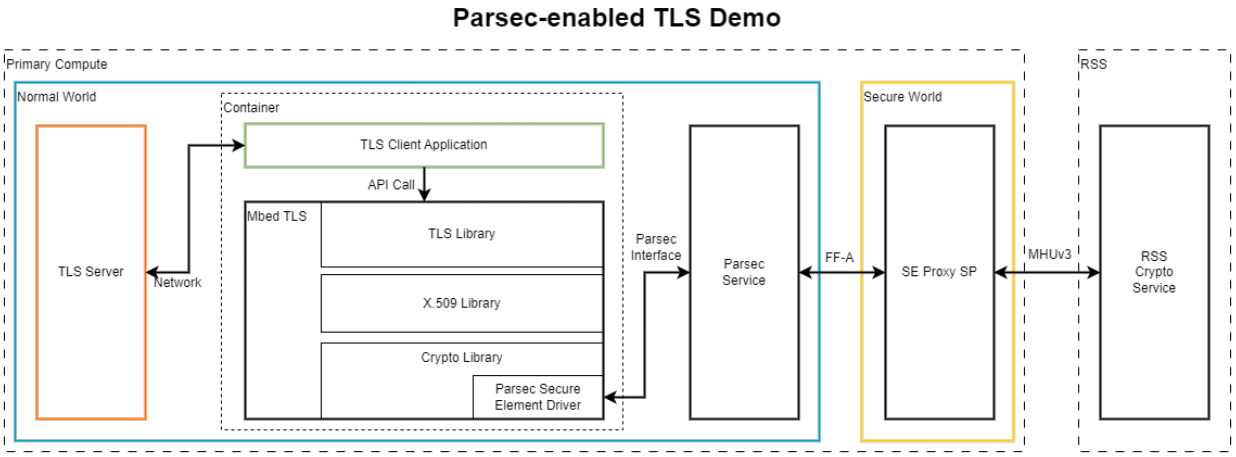
3.7.4 Parsec-enabled TLS Demo

Introduction

The Parsec-enabled TLS demo illustrates a HTTPS session where a Transport Layer Security (TLS) connection is established, and a simple webpage is transferred.

The TLS session consists of both symmetric and asymmetric cryptographic operations. In this demo, the symmetric operations are executed by Mbed TLS in Linux userspace. The asymmetric operations are carried out by Parsec. The backend of the Parsec service is based on the RSS crypto runtime service.

Architecture



Components

The following components are involved in the demo:

- TLS Server

  The TLS client application, running from inside a container, connects to the server at the `4433` port for the TLS connection. After the TLS connection is established, the server will send a simple `Hello` webpage when the client makes a request.

  The server application is provided by Mbed TLS. The source code can be found at `program/ssl/ssl_server.c` of Mbed TLS repository.

- TLS Client

The TLS client application connects to the server at the `4433` port for the TLS connection. It is deployed in a container environment. The client application calls the TLS API provided by Mbed TLS for the TLS connection.

The source code of the client application is based on the example program `program/ssl/ssl_client1.c` of Mbed TLS repository, with some modifications to handle a server IP address parameter and to work with Parsec. The code for modifications can be found at yocto/meta-kronos/recipes-demos/parsec/files.

- Mbed TLS

  Mbed TLS is a C library that implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and Datagram Transport Layer Security (DTLS) protocols.

  Mbed TLS provides 3 libraries:

  - TLS Library (libmbedcrypto)
  - X.509 Library (libmbedx509)
  - Crypto Library (libmbedcrypto)

  The relation of the 3 libraries is: libmbedtls depends on libmbedx509 and libmbedcrypto, and libmbedx509 depends on libmbedcrypto.

  In this demo, the client application code calls the API provided by the TLS Library of Mbed TLS to setup secure connection.

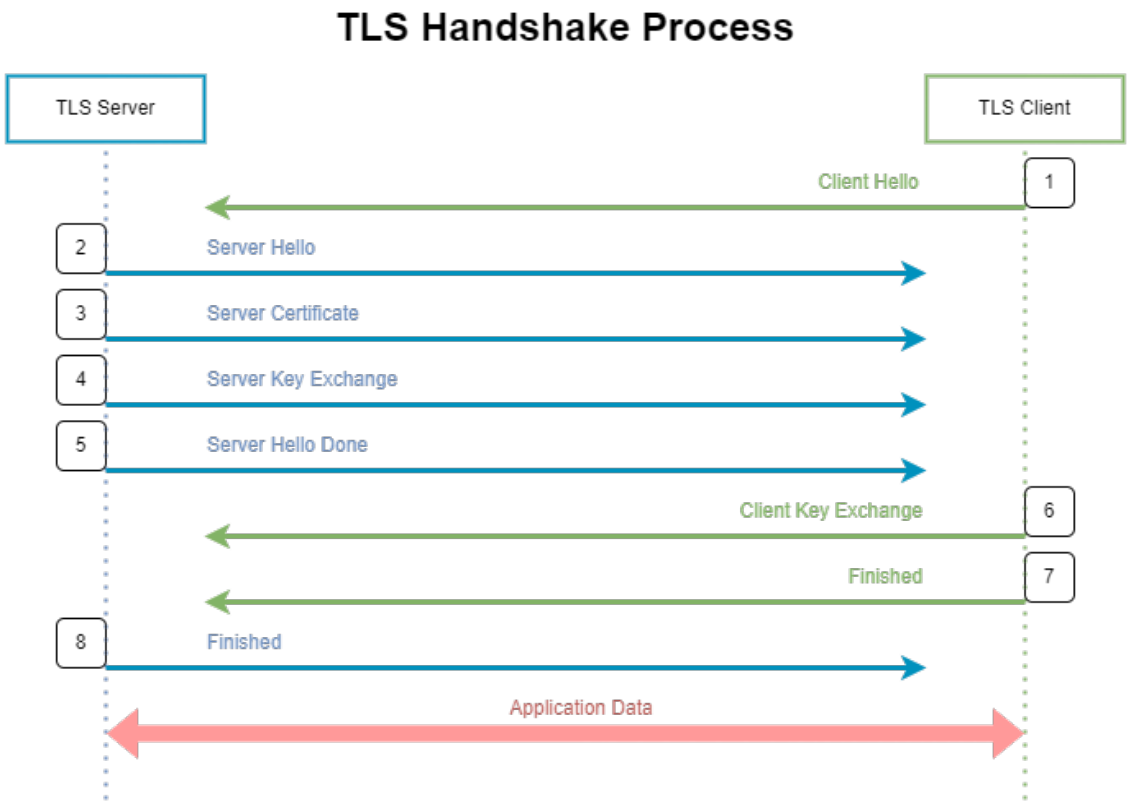
- Parsec Secure Element Driver

  The Parsec Secure Element Driver is an external driver for the Crypto Library of Mbed TLS. The driver implements a secure element by using the Parsec service. It delegates the crypto API calls to Parsec. The calls are further handled by the Secure Enclave Proxy Secure Partition (SE Proxy SP) in the Secure world of Primary Compute and finally handled by the RSS crypto service.

For more information of how the operations are handled by Parsec service, the SE Proxy SP and the RSS, refer to *Secure Services*.

TLS Handshake

The following diagram illustrates the TLS handshake process that happens in the demo.



To setup a TLS connection, the client and the server conduct the following main handshake steps:

1. **Client Hello**
   The client initiates the handshake by sending a "hello" message to the server.
2. **Server Hello**
   In reply to the client hello message, the server sends a "hello" message to the client.
3. **Server Certificate**
   The server transfers its certificate to the client. The client authenticates the server certificate.
4. **Server Key Exchange**
   This message conveys cryptographic information to allow the client to communicate the premaster secret (a 48-byte random number).
5. **Server Hello Done**
   The server finishes sending messages to support the key exchange and the client can proceed with its phase of the key exchange.
6. **Client Key Exchange**
   The client generates its own premaster secret, encrypts it with the server's public key obtained from the server certificate, and sends the encrypted data to the server.
7. **Client Finished**
   The client finishes the handshake.

8. **Server Finished**
   The server finishes the handshake.

Once the handshake finishes successfully, the server and the client can exchange data securely, because the data is encrypted with a symmetric algorithm.

Using RSS Crypto Service

In the TLS handshake step `3. Server Certificate` and `4. Server Key Exchange`, the client performs asymmetric crypto operations to verify digital signatures from the server side. The client invokes the Parsec Secure Element Driver in Mbed TLS to handle the asymmetric operations. Finally the operations are served by the RSS crypto runtime service. Specifically, the TLS client calls following APIs from the RSS for the asymmetric crypto operations:

- `psa_import_key`
  - The API imports a key in binary format. The TLS client application uses the API to import a public key.
- `psa_verify_hash`
  - The API verifies the signature of a hash or short message using a public key. The TLS client uses this API to verify digital signatures of the TLS server assets with the public key imported by `psa_import_key`.
- `psa_destroy_key`
  - The API destroys a key. The TLS client application destroys the public key imported by `psa_import_key`.

Validation

See *Integration Tests Validating the Parsec-enabled TLS Demo*.

3.7.5 Safety Island PSA Architecture Test Suite

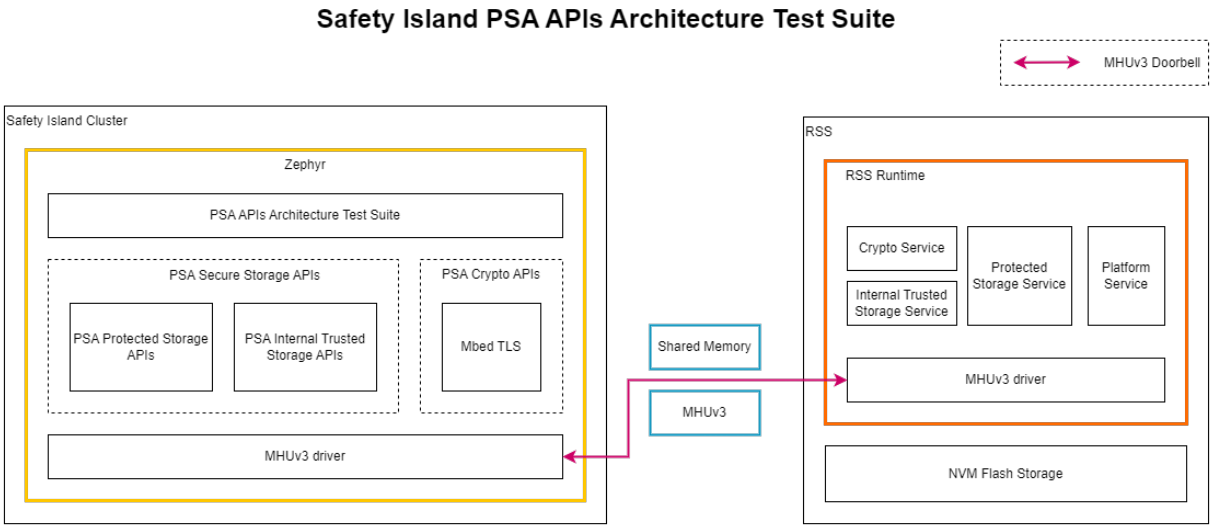
Introduction

The PSA Arch Tests test suite is one of a set of resources provided by Arm that can help organizations develop products that meet the security requirements of PSA Certified on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software.

The implementation of the PSA APIs Architecture Test Suite contains tests for PSA APIs specifications. The tests are available as open source.

The architecture test suite abstracts platform-specific information from the tests.

Diagram



Device Tree

In Zephyr, the device tree overlays the MHUv3 transmission and reception devices and defines the shared SRAM memory between the Safety Island cluster and RSS.

The Zephyr overlay device tree for FVP the Kronos board is located at components/safety_island/zephyr/src/overlays/psa.

PSA Secure Storage APIs Architecture Test Suite

The PSA Secure Storage APIs Architecture Test Suite runs on Safety Island Cluster 2 as a Zephyr application. It uses the PSA Secure Storage APIs interfaces provided by Trusted Firmware-M which communicates with the Secure Storage Service provided by the Trusted Firmware-M running on RSS using an RSS communication protocol.

The PSA Secure Storage API tests are linked into the Trusted Firmware-M PSA Secure Storage APIs binaries and will automatically run. A log similar to the following should be visible; it is normal for some tests to be skipped but there should be no failed tests:

```
***** PSA Architecture Test Suite - Version 1.4 *****
Running.. Storage Suite
****************************************
TEST: 401 | DESCRIPTION: UID not found check | UT: STORAGE
[Info] Executing tests from non-secure
[Info] Executing ITS tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
```

(continues on next page)

(continued from previous page)

```
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
Set storage for UID 6
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

[Info] Executing PS tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
Set storage for UID 6
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

TEST RESULT: PASSED

*****************************************

<further tests removed from log for brevity>

************ Storage Suite Report **********
TOTAL TESTS     : 17
TOTAL PASSED    : 11
TOTAL SIM ERROR : 0
TOTAL FAILED    : 0
TOTAL SKIPPED   : 6
*****************************************
```

There are some limitations behind running `PSA Secure Storage APIs Architecture Test Suite` on Safety Island Cluster 2 only. Refer to the release notes *Limitations* section.

PSA Secure Storage APIs

The PSA Secure Storage APIs are provided by the Trusted Firmware-M interfaces instead of duplicating code in Kronos Reference Software Stack. They are linked into Zephyr and use the provided `psa_call()` in order to communicate with the RSS to use the Secure Storage Service provided by Trusted Firmware-M.

Refer to Trusted Firmware-M PSA Protected Storage Interfaces and Trusted Firmware-M PSA Internal Trusted Storage Interfaces for more information.

PSA Crypto APIs Architecture Test Suite

The *PSA Crypto APIs Architecture Test Suite* is integrated in a Zephyr application. The application is deployed on all the 3 Safety Island Clusters.

The test suite contains 61 test cases in total. The test cases are executed in sequence. At the end of the test suite, a log similar to the following should be visible on all the 3 Safety Island terminals. Normally, no failure should be seen:

```
************ Crypto Suite Report **********
TOTAL TESTS     : 61
TOTAL PASSED    : 61
TOTAL SIM ERROR : 0
TOTAL FAILED    : 0
TOTAL SKIPPED   : 0
****************************************
```

PSA Crypto APIs

The PSA Crypto APIs are implemented by Mbed TLS. In Mbed TLS, different crypto APIs are handled in different ways. For asymmetric crypto operations, the RSS secure service is invoked by calling the `psa_call()` interface. The other crypto operations are handled on Safety Island by Mbed TLS software implementation. For more information on the Mbed TLS implementation, refer to *PSA Crypto APIs*.

Validation

See *Integration Tests Validating Safety Island PSA APIs Architecture Test Suite*.

Downstream Changes

Patch files can be found at yocto/meta-kronos/recipes-kernel/zephyr-kernel/files/psa-arch-tests to:

- Add PSA Arch Tests as a Zephyr module.
- Move a Secure Storage test to be the final one in the test suite as it causes Denial of Service to the Primary Compute.
- Change the key location of asymmetric crypto operation test cases, so the RSS secure service can be called.
- Postpone the time-consuming crypto test case for `psa_generate_key` to the end of the execution sequence.

3.8 Integration

The Reference Software Stack uses the Yocto Project build framework to build, integrate and validate the *Use-Cases*.

The Yocto Project version used by the Reference Software Stack is nanbield.

3.8.1 meta-kronos Yocto Layer

The `meta-kronos` layer primarily depends on the `meta-arm-bsp` layer which implements the `fvp-rd-kronos` bitbake `MACHINE` definition to enable the Reference Software Stack to run on the Arm Kronos Reference Design FVP (FVP_RD_Kronos). The layer `meta-kronos` is based on the Cassini distribution. It also contains a set of bitbake bbclasses, recipes and libraries to build, integrate, and validate the *Use-Cases* with either or both the **Baremetal** and **Virtualization** Reference Software Stack Architectures as described in *Reference Software Stack Overview*.

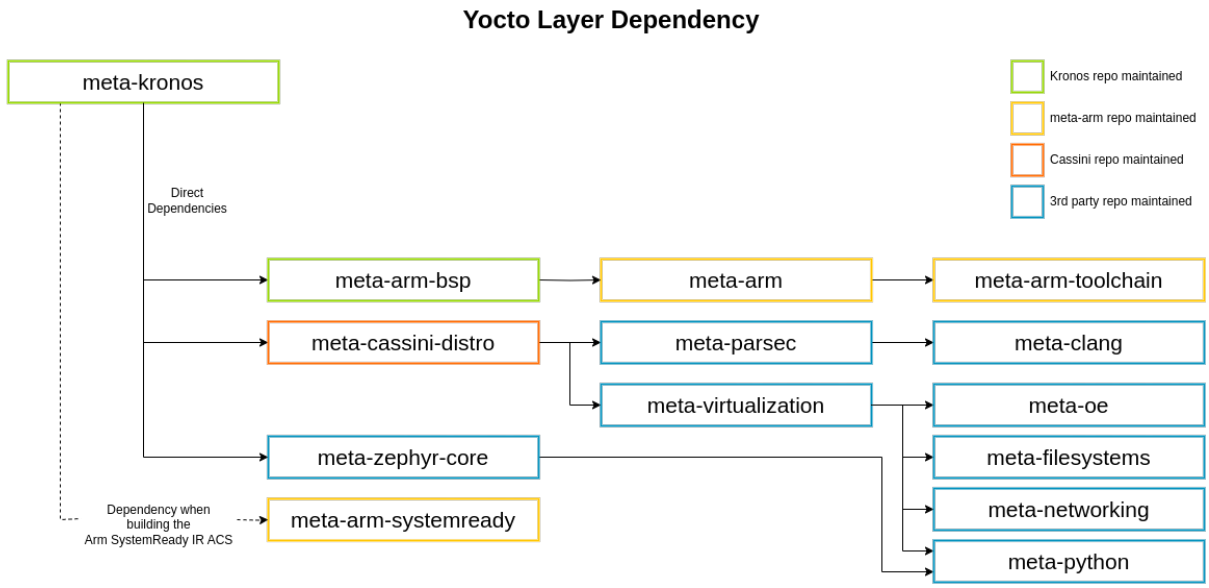
The layer source code can be found at yocto/meta-kronos.

Yocto Build Configuration

A set of `yaml` configuration files (found at yocto/kas) for the kas build tool is provided to support bitbake layer fetching, project configuration and executing the build and validation.

Yocto Layers Dependency

The following diagram illustrates the layers which are integrated as part of the Reference Software Stack.



Note that the `meta-arm-systemready` layer is only required when building for the Arm SystemReady IR ACS tests.

The layer dependency sources and their revisions for the `kronos` repository (https://gitlab.arm.com/automotive-and-industrial/kronos-ref-stack/kronos) v1.0 branch are:

```
URL: https://gitlab.arm.com/automotive-and-industrial/kronos-ref-stack/meta-arm
layers: meta-arm, meta-arm-bsp, meta-arm-systemready, meta-arm-toolchain
branch: kronos-nanbield
revision: 5e4851a884985b952b33f6f88a8724fbbe5300ec

URL: https://gitlab.com/Linaro/cassini/meta-cassini
layers: meta-cassini-distro
branch: nanbield
revision: v1.1.0

URL: https://github.com/kraj/meta-clang
layers: meta-clang
branch: nanbield
revision: 5170ec9cdfe215fcef146fa9142521bfad1d7d6c

URL: https://git.openembedded.org/meta-openembedded
layers: meta-filesystems, meta-networking, meta-oe, meta-python
branch: nanbield
revision: da9063bdfbe130f424ba487f167da68e0ce90e7d

URL: https://git.yoctoproject.org/git/meta-security
layers: meta-parsec
branch: nanbield
revision: 5938fa58396968cc6412b398d403e37da5b27fce

URL: https://git.yoctoproject.org/git/meta-virtualization
layers: meta-virtualization
branch: nanbield
revision: ac125d881f34ff356390e19e02964f8980d4ec38

URL: https://git.yoctoproject.org/git/meta-zephyr
layers: meta-zephyr-core
branch: nanbield
revision: fa76b75bd65da63abcc2d65dd5d4eb24296f2f65

URL: https://git.yoctoproject.org/git/poky
layers: meta, meta-poky
branch: nanbield
revision: 1a5c00f00c14cee3ba5d39c8c8db7a9738469eab
```

3.9 Validation

3.9.1 Run-Time Integration Tests

The run-time integration tests are a mechanism for validating the Reference Software Stack's core functionalities.

The tests are run on the image using the oeqa test framework. Refer to OEQA FVP for more information on this framework.

Note: There is a rare known failure where a timeout might occur during test execution. Refer to *Known Issues* for

possible workarounds.

In this section, details on the structure, implementation and debugging of the tests is given.

OEQA tests in meta-arm

The Processing Elements and Components tested by the framework are detailed below. The testing scripts can be found in meta-arm/lib/oeqa/runtime/cases/.

All of the Processing Elements and Components have their terminal output logged for debugging.

- **LCP**
  The script that implements the test is meta-arm/lib/oeqa/runtime/cases/test_00_lcp.py. The test waits for the LCP to log that it has successfully initialized and started all of its internal modules. It also checks whether the LCP has logged any errors, in which case the test fails.
- **RSS**
  The script that implements the test is meta-arm/lib/oeqa/runtime/cases/test_00_rss.py. The test firstly waits for the successful programming of the GIC-Multiview and the NI-710AE. Then the test waits for the RSS to log that it is releasing the SCP. This is its last action as part of the RSS boot process.
- **SCP**
  The script that implements the test is meta-arm/lib/oeqa/runtime/cases/test_00_scp.py. The test waits for the SCP to log that it has successfully initialized and started all of its internal modules. It also checks whether the SCP has logged any errors, in which case the test fails.
- **Primary Compute**
  - **BSP**
    The entry point to these tests is meta-arm/lib/oeqa/runtime/cases/fvp_devices.py. To find out more about the applicable tests, refer to *BSP Tests*.
  - **TF-A**
    The script that implements the test is meta-arm/lib/oeqa/runtime/cases/test_00_trusted_firmware_a.py. The test waits for the Primary Compute to log that it is entering the Normal world as defined in the RSS boot process.
  - **OP-TEE**
    The script that implements the test is meta-arm/lib/oeqa/runtime/cases/test_00_secure_partition.py. The test waits for the Primary Compute to log that OP-TEE loads the required SPs (Secure Partitions) and primary CPU switches to Normal world boot.

BSP Tests

The BSP Tests consist of a series of device tests that can be found in meta-arm/lib/oeqa/runtime/cases/fvp_devices.py.

- **networking**
  Checks that the network device and its correct driver are available and accessible via the filesystem and that outbound connections work (invoking `wget`).
- **rtc**
  Checks that the rtc (real-time clock) device and its correct driver are available and accessible via the filesystem and verifies that the `hwclock` command runs successfully.
- **cpu_hotplug**
  Checks for CPU availability and that basic functionality works, like enabling and stopping CPUs and preventing all of them from being disabled at the same time.

- **virtiorng**
  Check that the virtio-rng device is available through the filesystem and that it is able to generate random numbers when required.
- **watchdog**
  Checks that the watchdog device and its correct driver are available and accessible via the filesystem.

Integration Tests Implementation

This section gives a high-level description of how the integration testing logic is implemented.

To enable the integration tests, the testimage.bbclass is used. This class supports running automated tests against images. The class handles loading the tests and starting the image.

The Writing New Tests section of the Yocto Manual explains how to write new tests when using the testimage.bbclass. These are placed under meta-arm/lib/oeqa/runtime/cases and will be selected by the different machines/configurations by modifying the `TEST_SUITES` variable. For example, the file meta-arm-bsp/conf/machine/fvp-rd-kronos.conf adds the `fvp_devices` test to the `TEST_SUITES` variable.

Before running integration tests, some basic tests will be run firstly:

- **test_linux_login**
  The test logs in the Linux with root. If it fails, the tests that depend on it will be cancelled. The test is implemented in yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_linuxlogin.py.
- **test_cluster{N}**
  The test verifies the output when Zephyr boots on Safety Island Cluster N. The N is the Safety Island clusters number. Those tests are implemented in:

  yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_safety_island_c0.py

  yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_safety_island_c1.py

  yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_safety_island_c2.py

After running all the integration tests, the following test is run:

- **test_linux_shutdown**
  The test verifies that the FVP can be terminated using a `shutdown now` command in the linux console. The test is implemented in yocto/meta-kronos/lib/oeqa/runtime/cases/test_99_linuxshutdown.py.

Integration Tests Validating the Safety Island Actuation Demo

The `test_player_to_analyzer` integration test in yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_actuation.py does a full Player to Packet Analyzer functionality test.

This test invokes the Actuation Player that plays a recorded driving scenario which triggers the Actuation Service to generate Control Commands to be forwarded to the host via BSD socket. These Control Commands are then captured by the Packet Analyzer which validates them against a recorded Control Commands list that is stored in the form of a CSV file.

Integration Tests Validating the Critical Application Monitoring Demo

The script that implements the tests is yocto/meta-kronos/lib/oeqa/runtime/cases/test_40_cam.py.

The tests verify:

- The pack command of `cam-tool` on the Primary Compute.
- The stream data calibration on the Primary Compute.
- The startup of `cam-service` on the Safety Island Cluster 1.
- Application monitoring from the Safety Island Cluster 1.
- Application monitoring with multiple connections.
- Logical and temporal failure detection.

The tests are performed from the baremetal Linux userspace when building the Baremetal Architecture and from both the DomU1 and DomU2 Linux userspaces when building the Virtualization Architecture.

Integration Tests Validating the Safety Island Communication Demo

The scripts that implement the tests are yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_hipc.py and yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_hipc_virtualization.py. The tests below are run for each Safety Island cluster for Baremetal and Virtualization Architectures. For the Virtualization Architecture, tests are run for each Xen guests created.

- **test_ping_cluster**
  The test pings the Safety Island from the Primary Compute and vice versa and checks that an answer is received.
- **test_hipc_cluster**
  The test verifies Heterogeneous Inter Processor Communication (HIPC) between the Safety Island (using `zperf`) and the Primary Compute (using `iperf`). The tested configurations are:
  - The Safety Island as an iperf server (UDP/TCP) and the Primary Compute as a client (UDP/TCP).
  - The Safety Island as an iperf client (UDP/TCP) and the Primary Compute as a server (UDP/TCP).
- **test_hipc_cluster_cl{M}_cl{N}**
  The test verifies Heterogeneous Inter Processor Communication (HIPC) between the Safety Island Clusters (using `zperf`) where M and N are the clusters number. The tested configurations are:
  - The Safety Island Cluster {M} as an Zperf server (UDP/TCP) and the Safety Island Cluster {N} as a Zperf client (UDP/TCP).

Integration Tests Validating gPTP

The scripts that implement the tests are yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_ptp_base.py and yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_ptp.py.

- **test_ptp_linux_services**
  The test ensures the `ptp4l` services are running.
- **test_ptp_si_clients**
  The test verifies that the gPTP services running on the Safety Island clusters are in the expected client state. It then introduces a fault in the system by bringing down the relevant network interfaces on the server side and checks that the state machines on the Safety Island clusters are not in a client state anymore. The network interfaces are brought back up and the test validates that the state machines get back to the expected state.

- **test_ptp_domu_client**
  The test has the same steps as the Safety Island test above, but targeted at the DomUs instead of the Safety Island clusters. It is skipped when not using the Virtualization Architecture.

Integration Tests Validating the Safety Island Cluster 0 Bridge

The `test_si{N}_bridge_ethernet0` integration tests in yocto/meta-kronos/lib/oeqa/runtime/cases/test_30_si0_bridge_ethernet0.py verify the connection between the Host and the bridged Safety Island clusters. The tested configuration is:

- The Safety Island as an iperf server (TCP) and the Host as a client (TCP).

UDP is not tested because the user networking of the FVP does not provide port forwarding for UDP traffic.

Integration Tests Validating the Parsec-enabled TLS Demo

The `test_parsec_demo` integration test in yocto/meta-kronos/lib/oeqa/runtime/cases/test_40_parsec.py verifies the functionality of the crypto service provided by Parsec and the RSS. The test is only enabled when the use case is `Safety Island Actuation Demo` and the "Baremetal Architecture" is selected.

The test invokes a TLS server and client application. The client consumes the Parsec service for asymmetric crypto operations. Parsec is configured with Trusted Services as the backend which further invokes the crypto service from the hardware isolated RSS.

The test is performed under the following configuration:

- The TLS server operates on the Primary Compute.
- The TLS client operates within a container that resides on the Primary Compute.

Integration Tests Validating Xen

The `test_ptestrunner` integration test in yocto/meta-kronos/lib/oeqa/runtime/cases/test_40_virtualization.py uses `ptest-runner` to execute `01-xendomains.bats` BATS (Bash Automated Test System) tests in yocto/meta-kronos/recipes-test/xen/files/tests/01-xendomains.bats,

DomUs lifecycle management

The `01-xendomains.bats` BATS test verifies DomU lifecycle management, including status checking, destroy and restart.

MPAM

The MPAM cache partitioning functionality is verified by the `01-xendomains.bats` BATS tests from the following aspects:

- Verify if the Dom0 Cache Portion Bitmap (CPBM) value is consistent with the pre-set value from the Xen command line.
- Verify if the CPBM values for DomU1 and DomU2 are consistent with the pre-set value from the Xen guest configuration files.
- Verify if user can modify the domain CPBM values by the `xl` sub-commands in *MPAM*.

GICv4.1 vLPI/vSGI Direct Injection Demo

The `01-xendomains.bats` BATS test verifies GICv4.1 feature enablement, through pre-set keyword capture in Xen and Dom0 Linux boot log.

The `test_gicv4_1` integration test in yocto/meta-kronos/lib/oeqa/runtime/cases/test_40_gicv4_1.py verifies the functionality of GICv4.1 vLPI/vSGI direct injection. The test is only enabled when the use case is `Safety Island Actuation Demo` and the "Virtualization Architecture" is selected.

The test consists of the following aspects and will only be run on DomU1:

- PCI AHCI SATA disk `ahci[0000:00:1f.0]` has already been assigned to DomU1 with static PCI passthrough method.
- Using `lspci` command to check if PCI AHCI SATA disk is properly probed.
- Inspecting `/proc/interrupts` and reading non-zero MSI-X interrupts from `ahci[0000:00:00.0]` captured at domain boot-time to validate the functionality of GICv4.1 vLPI direct injection.
- Inspecting `/proc/interrupts` and reading non-zero IPI0 interrupts captured at domain boot-time to validate the functionality of GICv4.1 vSGI direct injection.

The testing of interrupt injection is not currently validated for run time operations, e.g. file system actions or data transfer.

Integration Tests Validating Primary Compute PSA APIs Architecture Test Suite

The meta-arm Yocto layer provides Trusted Service OEQA tests which can be used for automated Trusted Services Test Executables. The script that implements the test is meta-arm/lib/oeqa/runtime/cases/trusted_services.py.

Currently, only the following test cases for *psa-api-test* (from the PSA Arch Tests project) are supported:

- **ts-psa-crypto-api-test**
  Used for PSA API conformance testing for PSA Crypto API.
- **ts-psa-ps-api-test**
  Used for PSA API conformance testing for PSA Secure Storage API. Currently only the Protected Storage component of Secure Storage is supported.

Integration Tests Validating Safety Island PSA APIs Architecture Test Suite

The `test_psa_si_cluster{N}` integration tests in yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_si_psa_arch_tests.py verify that the psa-arch-tests suite report is as expected.

This test waits until the `psa-arch-tests` finish successfully and there are no failures in the tests suite report.

Integration Tests Validating the Fault Management Subsystem

The Fault Management test suite at yocto/meta-kronos/lib/oeqa/runtime/cases/test_10_fault_mgmt.py contains a test class to validate the FMUs and another to validate the SSUs using the Zephyr shell commands described in the *Shell Reference*.

`FaultMgmtTest` validates the configuration, injection, reporting and storage of faults in the FMU device tree:

- `test_tree` minimally validates the existence of the expected devices.
- `test_system_fmu_internal_inject` validates the injection and reporting of internal faults of the System FMU.

- `test_system_fmu_internal_set_enabled` validates disabling System FMU faults.
- `test_gic_fmu_inject` validates the injection and reporting of GIC-720AE FMU faults (critical and non-critical).
- `test_fmu_fault_count` validates the reporting of the overall fault count.
- `test_fmu_fault_list` validates the list of reported fault counts.
- `test_fmu_fault_summary` validates the fault summarization.
- `test_fmu_fault_clear` validates that injected faults can be cleared.

`FaultMgmtSSUTest` validates all possible transitions in the SSU state machine using three test cases (with a full system reset between each one to transition from `ERRC` back to `TEST`):

- `test_ssu_compl_ok`, which triggers a non-critical fault, recovers then triggers a critical fault.
- `test_ssu_nce_ok`, in which the self-test fails with a non-critical fault which is then signaled as critical.
- `test_ssu_ce_not_ok`, in which the self-test fails with a critical fault.

Integration Tests Validating SVE2

The script that implements the tests is yocto/meta-kronos/lib/oeqa/runtime/cases/test_40_sve.py

- **test_sve_enabled**
  The test ensures the `sve2` feature is enabled.
- **test_sve_config**
  This test verifies that the SVE2 configurations on the Primary Compute are valid in both virtualization and baremetal cases.

Integration Tests Validating Secure Firmware Update

The script that implements the tests is yocto/meta-kronos/lib/oeqa/runtime/cases/test_00_fwu.py

- **test_securefirmwareupdate**
  The test waits for U-Boot to start, starts the `Secure Firmware Update` process and ensures that the `Secure Firmware Update` was completed successfully by monitoring the RSS terminal output.

3.10 Arm SystemReady IR

Arm SystemReady is a compliance certification program based on a set of hardware and firmware standards that enable interoperability with generic off-the-shelf operating systems and hypervisors. These standards include the Base System Architecture (BSA) and Base Boot Requirements (BBR) specifications, and market-specific supplements.

In this way, the Arm SystemReady program provides a formal set of compute platform definitions to cover a range of systems from the cloud to IoT and edge, helping software 'just work' seamlessly across an ecosystem of Arm-based hardware.

Arm SystemReady is divided into a set of bands with a combination of specs available to suit the different devices and markets. Arm SystemReady IR is one of these bands.

Arm SystemReady IR certified platforms implement a minimum set of hardware and firmware features that an operating system can depend on to deploy the operating system image. Hence, Arm SystemReady IR ensures the deployment and maintenance of standard firmware interfaces and targets both custom (Yocto, OpenWRT, Buildroot) and pre-built (Debian, Fedora, SUSE) Linux distributions.

At a high level, the IR band requires that:

- Hardware implements the Base System Architecture (BSA)
- Firmware implements a subset of UEFI as defined in Embedded Base Boot Requirements (EBBR)
- Firmware by default provides a device tree suitable for booting mainline Linux
- Firmware can be updated using UEFI `UpdateCapsule()`
- At least three Linux distros must be able to boot, install, and run storage medium tests using the UEFI boot flow

Compliant systems must conform to the:

- Base System Architecture (BSA) specification
- Embedded Base Boot Requirements (EBBR)
- EBBR recipe of the Arm Base Boot Requirements (BBR) specification
- Device Tree specification
- Ethernet port requirements

It is also recommended to conform to the Security Interface Extension (SIE) certification. If that's not possible, the following Base Boot Security Requirements (BBSR) rules are still required:

- R140_BBSR: Capsule payloads for updating system firmware must be digitally signed
- R150_BBSR: Before updates to system firmware are applied, images must be verified using digital signatures

3.10.1 Support on Kronos Reference Software Stack

This Reference Software Stack aims to be aligned with Arm SystemReady IR version 2.1.0 by implementing most of its requirements. Given this is a reference design, the software is not being submitted for formal certification.

The support for running the Architectural Compliance Tests (ACS) is included in the Reference Software Stack. For more details on how to run it, refer to the *Arm SystemReady IR Architecture Compliance Suite (ACS) Tests* section of this documentation.

The Arm SystemReady scripts used to check the test results skip the identified non-alignments which are further described in the *Identified Non-Alignments* section below.

3.10.2 Identified Non-Alignments

The Reference Software Stack is currently known to have the following non-alignments:

- Reference Software Stack
  - Kronos system does not support populating the list of runtime variables, which will lead to "Can't populate EFI variables. No runtime variables will be available".
- Devicetree
  - Missing schemas for components which have not yet been or are not appropriate to be upstreamed (`arm,mhuv3`, `arm,mpam-msc`, `arm,rd-kronos`, `arm,slc`, `arm,si-channel`, `arm,si-rproc`).
- U-Boot
  - Known limitations of EFI implementation which are excluded in the EBBR Specification - UEFI Runtime Services.
  - Known limitations of EFI implementation which are noted as 'Explicit justification in a future revision of EBBR is pending' by edk2-test-parser.

  - The `UpdateCapsule()` method does not currently support certain invocations with invalid parameters.
- Model - FVP
  - Platform-specific limitations, which are noted as excluded in the EBBR Specification - Required Platform Specific Elements.
  - `AES`, `SHA1` and `SHA2` instructions are marked as unavailable in the FVP `ID_AA64ISAR0_EL1`.
- Test environment
  - No text input is available in the test environment for Simple Text Input Ex protocol.
- BSA tests
  - Tests are not compatible with certain devices in the RD-Kronos model.
- Distro installation
  - Only two Linux distro installations are performed (Debian and openSUSE), rather than the requisite three.

3.10.3 Arm SystemReady IR Tests

Arm SystemReady IR ACS Tests

The Arm SystemReady ACS (Architecture Compliance Suite) is a set of tests that ensure architectural compliance across different implementations and variants of the architecture. The ACS is delivered as a prebuilt release image. The image is a bootable live OS image containing a collection of test suites.

The meta-arm-systemready/classes/arm-systemready-acs.bbclass class in the meta-arm-systemready Yocto layer contains the common logic to deploy the Arm SystemReady IR ACS version 2.1.0 pre-built image and set up the testimage environment. It also contains a testimage "postfunc" called `acs_logs_handle` which generates report files and checks the results.

The script meta-arm-systemready/lib/oeqa/runtime/cases/arm_systemready_ir_acs.py in the meta-arm-systemready Yocto layer monitors the ACS tests output from the bitbake testimage task.

See meta-arm-systemready/README.md for more details. To run the tests, refer to *Arm SystemReady IR Architecture Compliance Suite (ACS) Tests*.

Linux Distributions Installation Tests

The Arm SystemReady IR requires that at least two Linux distros must be able to boot and install using the UEFI boot flow.

Recipes for testing the installation of Linux distributions are provided under meta-arm-systemready/recipes-test/arm-systemready-linux-distros. These recipes help to download the installation CD for the Linux distribution and generate an empty disk as the target disk for the installation.

See meta-arm-systemready/README.md for more details. To run the tests, refer to *Linux Distribution Installation (Debian and openSUSE)*.

CHAPTER

FOUR

LICENSE

The repository's standard license is the MIT license, under which most of the repository's content is provided (below).

```
Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice (including the next
paragraph) shall be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

Exceptions to this standard license relate to files that represent modifications to externally licensed works (for example, patch files). These files may therefore be included in the repository under alternative licenses in order to be compliant with the licensing requirements of the associated external works.

Find below the list of licenses related to externally licensed works:

- Apache-2.0
- BSD-2-Clause
- BSD-3-Clause
- BSD-3-Clause-Clear
- CC-BY-4.0
- GPL-2.0
- GPL-2.0-only
- LGPL-2.0
- LGPL-2.1

- MIT

4.1 SPDX Identifiers

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: MIT
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: http://spdx.org/licenses/

CHAPTER

FIVE

RELEASE NOTES

5.1 v1.0

5.1.1 New Features

Implementation of the *Use-Cases*.

The main components versions used in the Reference Software Stack:

Component	Version	Source
Kronos Reference Design FVP (FVP_RD_Kronos)	11.25.15	FVP download (arm64 host) FVP download (x86 host)
RSS (Trusted Firmware-M)	53aa78efef274b9e46e63b429078ae1863609728 (based on master branch post v1.8.1)	Trusted Firmware-M repository
SCP-firmware	cc4c9e017348d92054f74026ee1beb081403c168 (based on master branch post v2.13.0)	SCP-Firmware repository
Trusted Firmware-A	2.8.0	Trusted Firmware-A repository
OP-TEE	3.22.0	OP-TEE repository
Trusted Services	08b3d39471f4914186bd23793dc920e83b0e3197 (based on main branch, pre v1.0.0)	Trusted Services repository
U-Boot	2023.07.02	U-Boot repository
Xen	4.18	Xen repository
Linux Kernel	6.1.73	Linux repository and Linux preempt-rt repository
Zephyr	3.5.0	Zephyr repository
Safety Island Actuation Demo	v2.0	Actuation repository
Mbed TLS	1ec69067fa1351427f904362c1221b31538c8b57 (based on 3.5.0)	Mbed TLS repository
Critical Application Monitoring	v1.0	Critical Application Monitoring repository

Third-party Yocto layers used to build the Reference Software Stack:

```
URL: https://gitlab.arm.com/automotive-and-industrial/kronos-ref-stack/meta-arm
layers: meta-arm, meta-arm-bsp, meta-arm-systemready, meta-arm-toolchain
branch: kronos-nanbield
revision: 5e4851a884985b952b33f6f88a8724fbbe5300ec
```

(continues on next page)

(continued from previous page)

```
URL: https://gitlab.com/Linaro/cassini/meta-cassini
layers: meta-cassini-distro
branch: nanbield
revision: v1.1.0

URL: https://github.com/kraj/meta-clang
layers: meta-clang
branch: nanbield
revision: 5170ec9cdfe215fcef146fa9142521bfad1d7d6c

URL: https://git.openembedded.org/meta-openembedded
layers: meta-filesystems, meta-networking, meta-oe, meta-python
branch: nanbield
revision: da9063bdfbe130f424ba487f167da68e0ce90e7d

URL: https://git.yoctoproject.org/git/meta-security
layers: meta-parsec
branch: nanbield
revision: 5938fa58396968cc6412b398d403e37da5b27fce

URL: https://git.yoctoproject.org/git/meta-virtualization
layers: meta-virtualization
branch: nanbield
revision: ac125d881f34ff356390e19e02964f8980d4ec38

URL: https://git.yoctoproject.org/git/meta-zephyr
layers: meta-zephyr-core
branch: nanbield
revision: fa76b75bd65da63abcc2d65dd5d4eb24296f2f65

URL: https://git.yoctoproject.org/git/poky
layers: meta, meta-poky
branch: nanbield
revision: 1a5c00f00c14cee3ba5d39c8c8db7a9738469eab
```

5.1.2 Changed

Initial version.

5.1.3 Limitations

- In the HIPC, the iperf parameter "-l/–length" should be less than 1473 (IP and UDP overhead) in the case of Zephyr running as a UDP server since it does not support IP fragmentation.
- PSA Secure Storage API defines two interfaces for storages: Internal Trusted Storage (ITS) API and Protected Storage (PS) API. For now the Reference Software Stack supports the ITS API on Safety Island only.
- PSA Protected Storage Optional APIs `psa_ps_create` and `psa_ps_extended` are not supported by Kronos Reference Software Stack as they are not implemented in the Protected Storage Service provided by Trusted Firmware-M.

- PSA Secure Storage APIs Architecture Test Suite only runs on Cluster 2 in the Safety Island due to the following limitations:
  - Trusted Firmware-M supports a single partition only, this causes tests running simultaneously on different entities to interfere with each other due to accessing the same assets, resulting in failures.
  - Trusted Firmware-M has no support against Denial of Service attacks, where a test running on one entity might take up all the storage on the RSS resulting in denial of service for tests running on other entities.

5.1.4 Resolved and Known Issues

Known Issues

- The automated validation might fail due to the encoding issues in the logs. This has been observed on an AWS aarch64 Graviton 2 build host. On the test logs, the error message that appears is a typical timeout error.

  The console log appears normal, but some characters are either corrupted or replaced with 00, x00 or ^@ characters. This issue is likely caused by encoding mismatches or inconsistencies in the logging process, and it could occur in any of the test suites. A workaround is to trigger the "Automated Validation" again. When this issue occurs, something similar to the following would be observed in the logs:

```
52 28 bytes from 192.168.1.2 to 192.168.1.1: icmp_seq=7 ttl=64 time=0.00 ^@s^M
or
fault set_critical f\00u@2a570000 0x10000600 0
or
System shutdown complet\x00
```

- The automated validation might rarely fail due to timeouts related to the host CPU frequency and throttling, if this happens then simply running the automated validation again would fix such as issue.
- Refer to Critical Application Monitoring Known Issues for CAM-related known issues.

