Cassini

Arm Ltd.

Oct 04, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Project Cassini is the open, collaborative, standards-based initiative to deliver a seamless cloud-native software experience for devices based on Arm Cortex-A.

Initial release of Cassini distribution will provide a framework for deployment and orchestration of applications (edge runtime) within containers.

Future releases of Cassini distribution will include support for platform abstraction for security (PARSEC), provisioning the platform and update all components of software stack over the air. In addition, optionally utilize PARSEC to secure those operations.

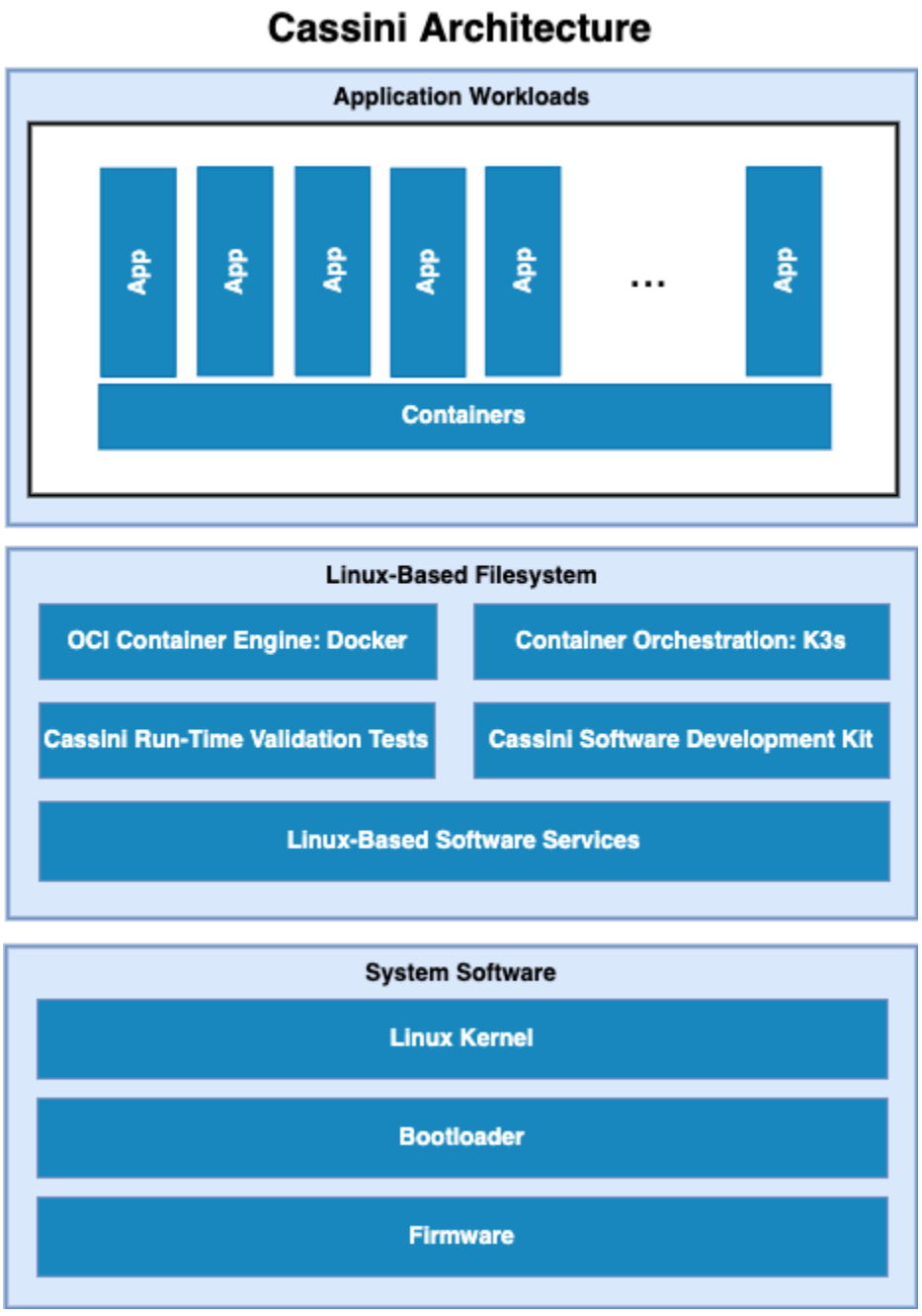
1.1 Use-Case Overview

Cassini aims to facilitate the deployment of application workloads via Docker and K3s use-case on the supported target platforms.

Instructions for achieving these use-cases are given in the *build* section, subject to relevant assumed technical knowledge as listed later in *documentation assumptions*.

1.2 Architecture

The following diagram illustrates the Cassini Architecture.



The different software layers are described below:

- **Application workloads**:

  User-defined container applications that are deployed and executed on the Cassini software stack. Note that the Cassini project provides the system infrastructure for user workloads, and not the application workloads themselves. Instead, they should be deployed by end-users according to their individual use-cases.

- **Linux-based filesystem**:

  This is the main component provided by the Cassini project. The Cassini filesystem contains tools and services that provide Cassini core functionalities and facilitate deployment and orchestration of user application workloads. These tools and services include the Docker container engine, the K3s container orchestration framework, together with their run-time dependencies. In addition, Cassini provides supporting packages such as those which

enable run-time validation tests or software development capabilities on the target platform.

- **System software**:

  System software specific to the target platform, composed of firmware, bootloader and the operating system.

1.3 Features Overview

Cassini includes the following major features:

- Container engine and runtime with Docker and runc-opencontainers.
- Container workload orchestration with the K3s Kubernetes distribution.
- On-target development support with optionally included Software Development Kit.
- Validation support with optionally included run-time integration tests, and build-time kernel configuration checks.

Other features of Cassini include:

- The features provided by the `poky.conf` distribution, which Cassini extends.
- Systemd used as the init system.
- RPM used as the package management system.

1.3.1 Documentation Assumptions

This documentation assumes a base level of knowledge related to different aspects of achieving the target use-case via Cassini:

- Application workload containerization, deployment, and orchestration

  This documentation does not provide detailed guidance on developing application workloads, deploying them, or managing their execution via Docker or the K3s orchestration framework, and instead focuses on Cassini-specific instructions to support these activities on an Cassini distribution image.

  For information on how to use these technologies which are provided with the Cassini distribution, see the Docker documentation and the K3s documentation.

- The Yocto Project

  This documentation contains instructions for achieving Cassini's use-case using a set of included configuration files that provide standard build features and settings. However, Cassini forms a distribution layer for integration with the Yocto project and is thus highly configurable and extensible. This documentation supports those activities by detailing the available options for Cassini-specific customizations and extensions, but assumes knowledge of the Yocto project necessary to prepare an appropriate build environment with these options configured.

  Readers are referred to the Yocto Project Documentation for information on setting up and running non-standard Cassini distribution builds.

1.4 Repository Structure

The `meta-cassini` repository is structured as follows:

- `meta-cassini`:
  - `meta-cassini-distro`

    Yocto distribution layer providing top-level and general policies for the Cassini distribution images.
  - `meta-cassini-tests`

    Yocto software layer with recipes that include run-time tests to validate Cassini functionalities.
  - `meta-cassini-config`

    Directory which contains configuration files for running tools on Cassini, such as files to support use of the kas build tool, or Cassini-specific configuration for running automated quality-assurance checks.

1.5 Repository License

The repository's standard licence 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.

Contributions to the project should follow the same licensing arrangement.

1.6 Contributions and Issue Reporting

Guidance for contributing to the Cassini project can be found at *Contributing*.

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

1.7 Feedback and support

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

1.8 Maintainer(s)

- Cassini Team

CHAPTER

TWO

USER MANUAL

2.1 Build, Deploy and validate Cassini Image

The recommended approach for image build setup and customization is to use the kas build tool. To support this, Cassini provides configuration files to setup and build different target images, different distribution image features, and set associated parameter configurations.

This page first briefly describes below the kas configuration files provided with Cassini, before guidance is given on using those kas configuration files to set up the Cassini distribution on a target platform.

Note: All command examples on this page can be copied by clicking the copy button. Any console prompts at the start of each line, comments, or empty lines will be automatically excluded from the copied text.

The `meta-cassini-config/kas` directory contains kas configuration files to support building and customizing Cassini distribution images via kas. These kas configuration files contain default parameter settings for a Cassini distribution build. Here, the files are briefly introduced, classified into three ordered categories:

- **Base Config**: Configures common software components
  - `cassini.yml` to prepare an image for the Cassini distribution.
- **Build Modifier Configs**: Set and configure features of the Cassini distribution
  - `tests.yml` to include run-time validation tests into the image.
  - `cassini-sdk.yml` to build an SDK image for the Cassini distribution.
  - `security.yml` to build a security-hardened Cassini distribution image.
- **Target Platform Configs**: Set the target platform

  Cassini currently supports the The Neoverse N1 System Development Platform (N1SDP), corresponding to the `n1sdp` `MACHINE` implemented in meta-arm-bsp. A single Target Platform Config is therefore provided:
  - `n1sdp.yml` to select the N1SDP as the target platform.

  To read documentation about the N1SDP, see the N1SDP Technical Reference Manual.

These kas configuration files can be used to build a custom Cassini distribution by passing one **Base Config**, zero or more **Build Modifier Configs**, and one **Target Platform Config** to the kas build tool, chained via a colon (:) character. Examples for this are given later in this document.

In the next section, guidance is provided for configuring, building and deploying Cassini distributions using these kas configuration files.

2.1.1 Build Host Environment Setup

This documentation assumes an Ubuntu-based Build Host, where the build steps have been validated on the Ubuntu 18.04.6 LTS Linux distribution.

A number of package dependencies must be installed on the Build Host to run build scenarios via the Yocto Project. The Yocto Project documentation provides the list of essential packages together with a command for their installation.

The recommended approach for building Cassini is to use the kas build tool. To install kas:

```
sudo -H pip3 install --upgrade kas==3.0.2
```

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

To deploy an Cassini distribution image onto the supported target platform, `bmap-tools` is used. This can be installed via:

```
sudo apt install bmap-tools
```

Note: The Build Host should have at least 65 GBytes of free disk space to build a Cassini distribution image.

2.1.2 Download

The `meta-cassini` repository can be downloaded using Git, via:

```
# Change the tag or branch to be fetched by replacing the value supplied to
# the --branch parameter option

git clone https://git.gitlab.arm.com/cassini/meta-cassini.git --branch v0.9.0
cd meta-cassini
```

2.1.3 Build

To build Cassini distribution image for the N1SDP hardware target platform:

```
kas build meta-cassini-config/kas/cassini.yml:meta-cassini-config/kas/n1sdp.yml
```

The resulting Cassini distribution image will be produced at: `build/tmp/deploy/images/n1sdp/cassini-image-base-n1sdp.*`

To build Cassini distribution image with the Cassini SDK for the N1SDP hardware target platform:

```
kas build meta-cassini-config/kas/cassini-sdk.yml:meta-cassini-config/kas/
↪n1sdp.yml
```

The resulting Cassini distribution image will be produced at: `build/tmp/deploy/images/n1sdp/cassini-image-sdk-image-n1sdp.*`

Cassini distribution images can be modified by adding run-time validation tests and security hardening to the distribution. This can be done by including `meta-cassini-config/kas/tests.yml` and `meta-cassini-config/kas/security.yml` kas configuration file as a Build Modifier.

2.1.4 Deploy

Instructions for deploying a Cassini distribution image on the supported N1SDP hardware target platform is divided into two parts:

- *Load the Image onto an USB Storage Device*
- *Update the N1SDP MCC Configuration MicroSD Card*

Note: As the image filenames vary depending on the base config and the SDK, the precise commands to deploy a Cassini distribution image vary. The following documentation denotes required instructions with sequentially numbered indexes (e.g., 1, 2, . . .), and distinguishes alternative instructions by denoting the alternatives alphabetically (e.g., A, B, . . .).

Load the Image onto an USB Storage Device

Cassini distribution images are produced as files with the `.wic.bmap` and `.wic.gz` extensions. They must first be loaded to the USB storage device, as follows:

1. Prepare the USB storage device (minimum size of 64 GB).

   Identify the USB storage device using `lsblk` command:

   ```
   lsblk
   ```

   This will output, for example:

   ```
   NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
   sdc      8:0    0    64G  0 disk
   ...
   ```

Warning: In this example, the USB storage device is the `/dev/sdc` device. As this may vary on different machines, care should be taken when copying and pasting the following commands.

2. Prepare for the image copy:

   ```
   sudo umount /dev/sdc*
   cd build/tmp/deploy/images/n1sdp/
   ```

Warning: The next step will result in all prior partitions and data on the USB storage device being erased. Please backup before continuing.

3. Flash the image onto the USB storage device using `bmap-tools`:

   A. Cassini distribution image:

   ```
   sudo bmaptool copy --bmap cassini-image-base-n1sdp.wic.bmap cassini-
   ↪image-base-n1sdp.wic.gz /dev/sdc
   ```

   B. Cassini-SDK distribution image:

```
sudo bmaptool copy --bmap cassini-image-sdk-n1sdp.wic.bmap cassini-
↪image-sdk-n1sdp.wic.gz /dev/sdc
```

The USB storage device can then be safely ejected from the Build Host, and plugged into one of the USB 3.0 ports on the N1SDP.

Update the N1SDP MCC Configuration MicroSD Card

Note: This process doesn't need to be performed every time the USB Storage Device gets updated. It is only necessary to update the MCC configuration microSD card when the Cassini major version changes.

This guidance requires a physical connection able to be established between the N1SDP and a PC that can be used to interface with it, here assumed to be the Build Host. The instructions are as follows:

1. Connect a USB-B cable between the Build Host and the DBG USB port of the N1SDP back panel.
2. Find four TTY USB devices in the `/dev` directory of the Build Host, via:

   ```
   ls /dev/ttyUSB*
   ```

   This will output, for example:

   ```
   /dev/ttyUSB0
   /dev/ttyUSB1
   /dev/ttyUSB2
   /dev/ttyUSB3
   ```

   By default the four ports are connected to the following devices:

   - ttyUSB<n> Motherboard Configuration Controller (MCC)
   - ttyUSB<n+1> Application processor (AP)
   - ttyUSB<n+2> System Control Processor (SCP)
   - ttyUSB<n+3> Manageability Control Processor (MCP)

   In this guide the ports are:

   - ttyUSB0: MCC
   - ttyUSB1: AP
   - ttyUSB2: SCP
   - ttyUSB3: MCP

   The ports are configured with the following settings:

   - 115200 Baud
   - 8N1
   - No hardware or software flow support

3. Connect to the N1SDP's MCC console. Any terminal applications such as `putty`, `screen` or `minicom` will work. The `screen` utility is used in the following command:

   ```
   sudo screen /dev/ttyUSB0 115200
   ```

4. Power-on the N1SDP via the power supply switch on the N1SDP tower. The MCC window will be shown. Type the following command at the Cmd> prompt to see MCC firmware version and a list of commands:

```
?
```

This will output, for example:

```
Arm N1SDP MCC Firmware v1.0.1
Build Date: Sep  5 2019
Build Time: 14:18:16
+ command ------------------+ function ---------------------------------+
| CAP "fname" [/A]          | captures serial data to a file            |
|                           |  [/A option appends data to a file]       |
| FILL "fname" [nnnn]       | create a file filled with text            |
|                           |  [nnnn - number of lines, default=1000]   |
| TYPE "fname"              | displays the content of a text file       |
| REN "fname1" "fname2"     | renames a file 'fname1' to 'fname2'       |
| COPY "fin" ["fin2"] "fout"| copies a file 'fin' to 'fout' file        |
|                           |  ['fin2' option merges 'fin' and 'fin2']  |
| DEL "fname"               | deletes a file                            |
| DIR "[mask]"              | displays a list of files in the directory |
| FORMAT [label]            | formats Flash Memory Card                 |
| USB_ON                    | Enable usb                                |
| USB_OFF                   | Disable usb                               |
| SHUTDOWN                  | Shutdown PSU (leave micro running)        |
| REBOOT                    | Power cycle system and reboot             |
| RESET                     | Reset Board using CB_nRST                 |
| DEBUG                     | Enters debug menu                         |
| EEPROM                    | Enters eeprom menu                        |
| HELP  or  ?               | displays this help                        |
|                                                                       |
| THE FOLLOWING COMMANDS ARE ONLY AVAILABLE IN RUN MODE                 |
|                                                                       |
| CASE_FAN_SPEED "SPEED"    | Choose from SLOW, MEDIUM, FAST            |
| READ_AXI "fname"          | Read system memory to file 'fname'        |
|          "address"        | from address to end address               |
|          "end_address"    |                                           |
| WRITE_AXI "fname"         | Write file 'fname' to system memory       |
|           "address"       | at address                                |
+---------------------------+-------------------------------------------+
```

5. In the MCC window at the Cmd> prompt, enable USB via:

```
USB_ON
```

6. Mount the N1SDP's internal microSD card over the DBG USB connection to the Build Host and copy the required files.

   The microSD card is visible on the Build Host as a disk device after issuing the USB_ON command in the MCC console, as performed in the previous step. This can be found using the `lsblk` command:

```
lsblk
```

This will output, for example:

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sdb      8:0    0     2G  0 disk
└─sdb1   8:1    0     2G  0 part
```

Warning: In this example, the `/dev/sdb1` partition is being mounted. As this may vary on different machines, care should be taken when copying and pasting the following commands.

Mount the device and check its contents:

```
sudo umount /dev/sdb1
sudo mkdir -p /tmp/sdcard
sudo mount /dev/sdb1 /tmp/sdcard
ls /tmp/sdcard
```

This should output, for example:

```
config.txt   ee0316a.txt   LICENSES   LOG.TXT   MB   SOFTWARE
```

7. Wipe the mounted microSD card, then extract the contents of `n1sdp-board-firmware_primary.tar.gz` onto it:

```
sudo rm -rf /tmp/sdcard/*
sudo tar --no-same-owner -xf \
  build/tmp/deploy/images/n1sdp/n1sdp-board-firmware_primary.tar.gz -C \
  /tmp/sdcard/ && sync
sudo umount /tmp/sdcard
sudo rmdir /tmp/sdcard
```

Note: If the N1SDP board was manufactured after November 2019 (Serial Number greater than `36253xxx`), a different PMIC firmware image must be used to prevent potential damage to the board. More details can be found in Potential firmware damage notice. The `MB/HBI0316A/io_v123f.txt` file located in the microSD needs to be updated. To update it, set the PMIC image (`300k_8c2.bin`) to be used in the newer models by running the following commands on the Build Host:

```
sudo umount /dev/sdb1
sudo mkdir -p /tmp/sdcard
sudo mount /dev/sdb1 /tmp/sdcard
sudo sed -i '/^MBPMIC: pms_0V85.bin/s/^/;/g' /tmp/sdcard/MB/HBI0316A/io_v123f.
↪txt
sudo sed -i '/^;MBPMIC: 300k_8c2.bin/s/^;//g' /tmp/sdcard/MB/HBI0316A/io_v123f.
↪txt
sudo umount /tmp/sdcard
sudo rmdir /tmp/sdcard
```

2.1.5 Run

To run the deployed Cassini distribution image, simply boot the target platform. For example, on the MCC console accessed via the connected machine described in *Deploy*, reset the target platform and boot into the deployed Cassini distribution image via:

```
REBOOT
```

The resulting Cassini distribution image can be logged into as `cassini` user.

The distribution can then be used for deployment and orchestration of application workloads in order to achieve the desired use-cases.

2.1.6 Validate

As an initial validation step, check that the appropriate Systemd services are running successfully,

- `docker.service`
- `k3s.service`

These services can be checked by running the command:

```
systemctl status --no-pager --lines=0 docker.service k3s.service
```

And ensuring the command output lists them as active and running.

More thorough run-time validation of Cassini components are provided as a series of integration tests, available if the `meta-cassini-config/kas/tests.yml` kas configuration file was included in the image build.

2.1.7 Reproducing the Cassini Use-Cases

This section briefly demonstrates simplified use-case examples, where detailed instructions for developing, deploying, and orchestrating application workloads are left to the external documentation of the relevant technology.

Deploying Application Workloads via Docker and K3s

This example deploys the Nginx webserver as an application workload, using the `nginx` container image available from Docker's default image repository. The deployment can be achieved either via Docker or via K3s, as follows:

1. Boot the image and log-in as `cassini` user.
2. Deploy the example application workload:
   - **Deploy via Docker**

     2.1. Run the following example command to deploy via Docker:

     ```
     sudo docker run -p 8082:80 -d nginx
     ```

     2.2. Confirm the Docker container is running by checking its `STATUS` in the container list:

     ```
     sudo docker container list
     ```

   - **Deploy via K3s**

     2.1. Run the following example command to deploy via K3s:

```
cat << EOT > nginx-example.yml && sudo kubectl apply -f nginx-example.yml
apiVersion: v1
kind: Pod
metadata:
  name: k3s-nginx-example
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
      hostPort: 8082
EOT
```

2.2. Confirm that the K3s Pod hosting the container is running by checking that its `STATUS` is `running`, using:

```
sudo kubectl get pods -o wide
```

3. After the Nginx application workload has been successfully deployed, it can be interacted with on the network, via for example:

```
wget localhost:8082
```

Note: As both methods deploy a webserver listening on port 8082, the two methods cannot be run simultaneously and one deployment must be stopped before the other can start.

CHAPTER

THREE

DEVELOPER MANUAL

3.1 User Accounts

Cassini distribution images contain the following user accounts:

- `root` with administrative privileges enabled by default. The login is disabled if `cassini-security` is included in `DISTRO_FEATURES`.

  > **Note:** When `cassini-test` distro feature is enabled then `root` login is enabled. Currently, running `inline tests` in LAVA require login as `root` to run transfer-overlay commands.

- `cassini` with administrative privileges enabled with `sudo`.
- `user` without administrative privileges.
- `test` with administrative privileges enabled with `sudo`. This account is created only if `cassini-test` is included in `DISTRO_FEATURES`.

By default, each users account has disabled password. The default administrative group name is `sudo`. Other sudoers configuration is included in `meta-cassini-distro/recipes-extended/sudo/files/cassini_admin_group.in`.

If `cassini-security` is included in `DISTRO_FEATURES`, each user is prompted to a set new password on first login. For more information about security see: *security hardening*.

All *Run-Time Integration Tests* are executed as the `test` user.

A Cassini distribution image can be configured to include run-time integration tests that validate successful configuration of the Cassini user accounts. Details of the user accounts validation tests can be found in the *User Accounts Tests* section of the *Validation* documentation.

3.2 Build System

A Cassini distribution can be built by setting the target platform via the `MACHINE` BitBake variable. In addition, the desired distribution features via the `DISTRO_FEATURES` BitBake variable. Finally, customizing those features via feature-specific modifiable variables, if needed.

This chapter provides an overview of Cassini's support for the kas build tool. All the available distribution image features and supported target platforms are defined together with their associated kas configuration files, followed by any other additional customization options. The process for building without kas is then briefly described.

3.2.1 kas Build Tool Support

The kas build tool enables automatic fetch and inclusion of layer sources, as well as parameter and feature specifications for building target images. To enable this, kas configuration files in the YAML format are passed to the tool to provide the necessary definitions.

These kas configuration files are modular, where passing multiple files will result in an image produced with their combined configuration. Further, kas configuration files can extend other kas configuration files, thereby enabling specialized configurations that inherit common configurations.

The `meta-cassini-config/kas` directory contains kas configuration files that support building images via kas for the Cassini project, and fall into three ordered categories:

- **Base Config**
- **Build Modifier Configs**
- **Target Platform Configs**

To build an Cassini distribution image via kas, it is required to provide the **Base Config** and one **Target Platform Config**, unless otherwise stated in their descriptions below. Additional **Build Modifier Configs** are optional, and depend on the target use-case. Currently, it is necessary that kas configuration files are provided in order: The **Base Config** and then additional build features via zero or more **Build Modifier Configs**, and finally the **Target Platform Config**.

To enable builds for a supported target platform or configure each Cassini distribution image feature, kas configurations files are described in their relevant sections below: *Target Platforms* and *Distribution Image Features*, respectively. Example usage of these kas configuration files can be found in the *Build* section of the User Manual.

Note: If a kas configuration file does not set a particular build parameter, the parameter will take its default value.

3.2.2 Target Platforms

N1SDP

- **Corresponding value for `MACHINE` variable**: `n1sdp`.
- **Target Platform Config**: `meta-cassini-config/kas/n1sdp.yml`.

This supported target platform is the Neoverse N1 System Development Platform (N1SDP), implemented in meta-arm-bsp.

The `n1sdp.yml` **Target Platform Config** includes common configuration from the `meta-cassini-config/kas/include/arm-machines.yml` which defines the BSPs, layers, and dependencies required when building for the platform.

3.2.3 Distribution Image Features

For a particular target platform, the available Cassini distribution image features (corresponding to the contents of the `DISTRO_FEATURES` BitBake variable) are detailed in this section, along with any associated kas configuration files, and any associated customization options relevant for that feature.

Cassini Architecture

Cassini distribution image can be configured via kas using **Base Config**. This includes a set of common configuration from a base Cassini kas configuration file:

- `meta-cassini-config/kas/include/cassini-base.yml`

This kas configuration file defines the base Cassini layer dependencies and their software sources, as well as additional build configuration variables. It also includes the `meta-cassini-config/kas/include/cassini-release.yml` kas configuration file, where the layers dependencies are pinned for any corresponding Cassini release.

- **Corresponding value in `DISTRO` variable**: `cassini`.
- **Base Config**: `meta-cassini-config/kas/cassini.yml`.

This Cassini distribution image feature enables the `cassini-image-base` build target, to build an Cassini distribution image.

The **Base Config** for this distribution image feature sets the build target to `cassini-image-base`.

To build Cassini distribution image, provide the **Base Config** to the kas build command. For example, to build a Cassini distribution image for the N1SDP hardware target platform, run the following command:

```
kas build meta-cassini-config/kas/cassini.yml:meta-cassini-config/kas/
↪n1sdp.yml
```

Other Cassini Features

Developer Support

Corresponding value in `DISTRO_FEATURES` variable: `cassini-dev`.

This Cassini distribution image feature includes packages appropriate for development image builds, such as the `debug-tweaks` package, which sets an empty root password for simplified development access.

Run-Time Integration Tests

- **Corresponding value in `DISTRO_FEATURES` variable**: `cassini-test`.
- **Build Modifier Config**: `meta-cassini-config/kas/tests.yml`.

This Cassini distribution image feature includes the Cassini test suites provided to validate the image is running successfully with the expected Cassini functionalities.

The Build Modifier for this distribution image feature automatically includes the Yocto Package Test (ptest) framework in the image, configures the inclusion of `meta-cassini-tests` as a Yocto layer source for the build, and appends the `cassini-test` feature to `DISTRO_FEATURES` for the build.

To include run-time integration tests in a Cassini distribution image, provide the **Build Modifier Config** to the kas build command. For example, to include the tests in a Cassini distribution image for the N1SDP hardware target platform, run the following command:

```
kas build meta-cassini-config/kas/cassini.yml:meta-cassini-config/kas/tests.
↪yml:meta-cassini-config/kas/n1sdp.yml
```

Each suite of run-time integration tests and specific customizable variables associated with each suite are detailed separately, at *Run-Time Integration Tests*.

Security Hardening

- **Corresponding value in** `DISTRO_FEATURES` **variable**: `cassini-security`.
- **Build Modifier Config**: `meta-cassini-config/kas/security.yml`.

  This Cassini distribution image feature configures user accounts, packages, remote access controls and other image features to provide extra security hardening for the Cassini distribution image.

  To include extra security hardening in a Cassini distribution image, provide the **Build Modifier Config** to the kas build command, which appends the `cassini-security` feature to `DISTRO_FEATURES` for the build. For example, to include it in the Cassini distribution image for the N1SDP hardware target platform, run the following command:

```
kas build meta-cassini-config/kas/cassini.yml:meta-cassini-config/kas/security.
↪yml:meta-cassini-config/kas/n1sdp.yml
```

  The security hardening is described in more detail at *Security Hardening*.

Software Development Kit (SDK)

- **Corresponding value in** `DISTRO_FEATURES` **variable**: `cassini-sdk`.
- **Build Modifier Config**: `meta-cassini-config/kas/cassini-sdk.yml`

  This Cassini distribution image feature:

  - Adds the Cassini Software Development Kit (SDK) which includes packages and image features to support on-target software development activities.
  - Enables an additional SDK build target, `cassini-image-sdk`

  The Build Modifier for this distribution image feature automatically appends `cassini-sdk` to `DISTRO_FEATURES`, and sets the appropriate build target with the necessary configuration included by default.

  To include the SDK in a Cassini distribution image, provide the appropriate SDK **Build Modifier Config** to the kas build command. For example, to include the SDK in a Cassini distribution image for the N1SDP hardware target platform, run the following command:

```
kas build meta-cassini-config/kas/baremetal-sdk.yml:meta-cassini-config/kas/n1sdp.
↪yml
```

  The SDK itself is described in more detail at *Software Development Kit (SDK)*.

3.2.4 Additional Distribution Image Customizations

An additional set of customization options are available for Cassini distribution images, which don't fall under a distinct distribution image feature. These customizations are listed below and are grouped by the customization target.

Filesystem Customization

Adding Extra Rootfs Space

The size of the root filesystem can be extended via the `CASSINI_ROOTFS_EXTRA_SPACE` BitBake variable, which defaults to `2000000` Kilobytes. The value of this variable is appended to the `IMAGE_ROOTFS_EXTRA_SPACE` BitBake variable.

Tuning the Filesystem Compilation

The Cassini filesystem by default uses the generic `armv8a-crc` tune for `aarch64` based target platforms. This reduces build times by increasing the sstate-cache reused between different image types and target platforms. This optimization can be disabled by setting `CASSINI_GENERIC_ARM64_FILESYSTEM` to `"0"`. The file system compilation tune used when `CASSINI_GENERIC_ARM64_FILESYSTEM` is enabled can be changed by setting `CASSINI_GENERIC_ARM64_DEFAULTTUNE`, which configures the `DEFAULTTUNE` BitBake variable for the `aarch64` based target platforms builds. See DEFAULTTUNE for more information.

In summary, the relevant variables and their default values are:

```
CASSINI_GENERIC_ARM64_FILESYSTEM: "1"              # Enable generic file system␣
↪(1 or 0).
CASSINI_GENERIC_ARM64_DEFAULTTUNE: "armv8a-crc"    # Value of DEFAULTTUNE if␣
↪generic file system enabled.
```

Their values can be set by passing them as environmental variables. For example, the optimization can be disabled using:

```
CASSINI_GENERIC_ARM64_FILESYSTEM="0" kas build meta-cassini-config/kas/cassini.
↪yml:meta-cassini-config/kas/n1sdp.yml
```

3.2.5 Manual BitBake Build Setup

In order to build an Cassini distribution image without the kas build tool directly via BitBake, it is necessary to prepare a BitBake project as follows:

- Configure *dependent Yocto layers* in `bblayers.conf`.
- Configure the `DISTRO` as `cassini` in `local.conf`.
- Configure the image `DISTRO_FEATURES` in `local.conf`.

Assuming correct environment configuration, the BitBake build can then be run for the desired image target corresponding to one of the following:

- `cassini-image-base`
- `cassini-image-sdk`

As the kas build configuration files within the `meta-cassini-config/kas/` directory define the recommended build settings for each feature. Any additional functionalities may therefore be enabled by reading these configuration files and manually inserting their changes into the BitBake build environment.

3.3 Yocto Layers

The `meta-cassini` repository provides two layers compatible with the Yocto Project, in the following sub-directories:

- `meta-cassini-distro`

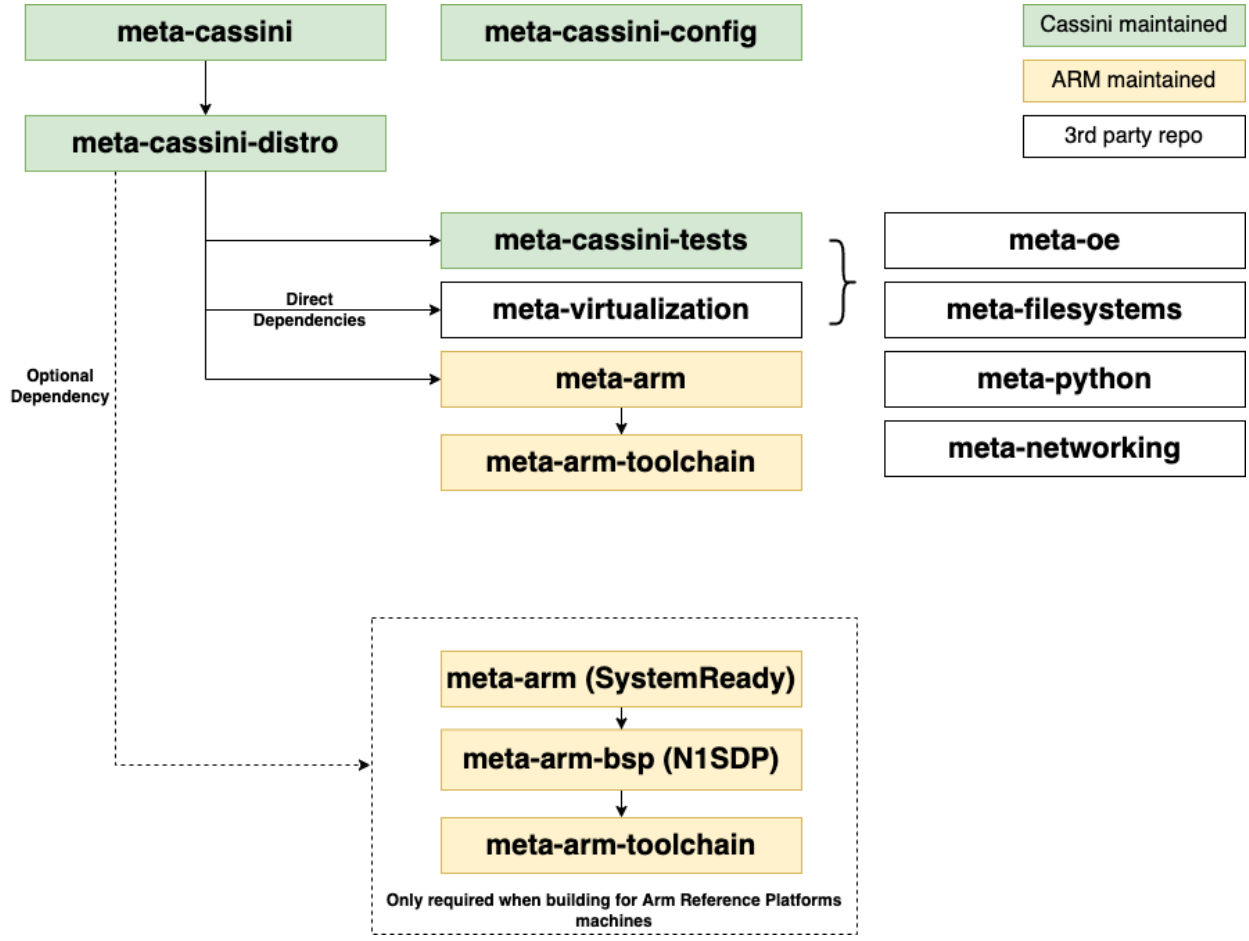
  Yocto distribution layer providing top-level and general policies for the Cassini distribution images.

- `meta-cassini-tests`

  Yocto software layer with recipes that include run-time tests to validate Cassini functionalities.

3.3.1 Layer Dependency Overview

The following diagram illustrates the layers which are integrated by the Cassini project, which are further expanded on below. The layer revisions are related to the Cassini `v0.9.0` release.



Cassini depends on the following layer dependency sources:

```
URL: https://git.yoctoproject.org/git/poky
layers: meta, meta-poky
branch: kirkstone
revision: 387ab5f18b17c3af3e9e30dc58584641a70f359f

URL: https://git.openembedded.org/meta-openembedded
layers: meta-filesystems, meta-networking, meta-oe, meta-python
branch: kirkstone
revision: 8f2dc1023482863e2630d1b94052c41ce748b38f

URL: https://git.yoctoproject.org/git/meta-virtualization
layer: meta-virtualization
branch: kirkstone
revision: 26a361a39ff5ab6fae22efbdc582f84d13330ba2
```

An additional layer dependency source is conditionally required, depending on the specific Cassini distribution image being built. This layer dependency source is the `meta-arm` repository, which provides three Yocto layers:

```
URL: https://git.yoctoproject.org/git/meta-arm
layers: meta-arm, meta-arm-bsp, meta-arm-toolchain
branch: kirkstone
revision: cf9365fcec2e741c56ad88db7f3838f636e29cae
```

The layers required from `meta-arm` depend on the Cassini distribution image:

- Cassini SDK distribution images require `meta-arm` and `meta-arm-toolchain`, as the `gator-daemon` package is installed on the rootfs.
- A Cassini distribution image built for the N1SDP hardware target platform requires `meta-arm`, `meta-arm-bsp`, and `meta-arm-toolchain`.

These layers are described as follows:

- `meta-arm`:
  - URL: https://git.yoctoproject.org/cgit/cgit.cgi/meta-arm/tree/meta-arm.
  - Clean separation between Firmware and OS.
  - The canonical source for SystemReady firmware.
- `meta-arm-bsp`:
  - URL: https://git.yoctoproject.org/cgit/cgit.cgi/meta-arm/tree/meta-arm-bsp.
  - Board specific components for Arm target platforms.
- `meta-arm-toolchain`:
  - URL: https://git.yoctoproject.org/meta-arm/tree/meta-arm-toolchain.
  - Provides toolchain for Arm target platforms

3.4 Security Hardening

Cassini distribution images can be hardened to reduce potential sources or attack vectors of security vulnerabilities. Cassini security hardening modifies the distribution to:

- Force password update for each user account after first logging in. An empty and expired password is set for each user account by default.
- Enhance the kernel security, kernel configuration is extended with the `security.scc` in `KERNEL_FEATURES`.
- Enable the 'Secure Computing Mode' (seccomp) Linux kernel feature by appending `seccomp` to `DISTRO_FEATURES`.
- Ensure that all available packages from `meta-openembedded` and `poky` layers are configured with: `--with-libcap[-ng]`.
- Remove `debug-tweaks` from `IMAGE_FEATURES`.
- Disable all login access to the `root` account.

Note: When `cassini-test` distro feature is enabled then `root` login is enabled. Currently, running `inline tests` in LAVA require login as `root` to run transfer-overlay commands.

- Sets the umask to `0027` (which translates permissions as `640` for files and `750` for directories).

Security hardening is not enabled by default, see *Security Hardening* for details on including the security hardening on the Cassini distribution image.

Note: Cassini security hardening does not reduce the scope of the *Run-Time Integration Tests*.

3.5 Software Development Kit (SDK)

Cassini SDK distribution images enable users to perform common development tasks on the target, such as:

- Application and kernel module compilation
- Remote debugging
- Profiling
- Tracing
- Runtime package management

The precise list of packages and image features provided as part of the Cassini SDK can be found in `meta-cassini-distro/conf/distro/include/cassini-sdk.inc`.

The Yocto project provides guidance for some of these common development tasks, for example kernel module compilation, profiling and tracing, and runtime package management.

See *Software Development Kit (SDK)* for details on including the SDK on a Cassini distribution image.

3.6 Validation

3.6.1 Build-Time Kernel Configuration Check

After the kernel configuration has been produced during the build, it is checked to validate the presence of necessary kernel configuration to comply with specific Cassini functionalities.

A list of required kernel configs is used as a reference, and compared against the list of available configs in the kernel build. All reference configs need to be present either as module (`=m`) or built-in (`=y`). A BitBake warning message is produced if the kernel is not configured as expected.

The following kernel configuration checks are performed:

- **Container engine support**:

  Check performed via: `meta-cassini-distro/classes/containers_kernelcfg_check.bbclass`. By default Yocto Docker config is used as the reference.

- **K3s orchestration support**:

  Check performed via: `meta-cassini-distro/classes/k3s_kernelcfg_check.bbclass`. By default Yocto K3s config is used as the reference.

3.6.2 Run-Time Integration Tests

The `meta-cassini-tests` Yocto layer contains recipes and configuration for including run-time integration tests into an Cassini distribution, to be run manually after booting the image.

The Cassini run-time integration tests are a mechanism for validating Cassini core functionalities. The following integration test suites are included in the Cassini distribution image:

- *Container Engine Tests*
- *K3s Orchestration Tests* (local deployment of a K3s pod)
- *User Accounts Tests*

The tests are built as a Yocto Package Test (ptest), and implemented using the Bash Automated Test System (BATS).

Run-time integration tests are not included in a Cassini distribution image by default, and must instead be included explicitly. See *Run-Time Integration Tests* within the Build System documentation for details on how to include the tests.

The test suites are executed using the `test` user account, which has `sudo` privileges. More information about user accounts can be found at *User Accounts*.

Running the Tests

If the tests have been included in the Cassini distribution image, they may be run via the ptest framework, using the following command after booting the image and logging in:

```
ptest-runner [test-suite-id]
```

If the test suite identifier (`[test-suite-id]`) is omitted, all integration tests will be run. For example, running `ptest-runner` produces output such as the following:

```
$ ptest-runner
START: ptest-runner
[...]
PASS:container-engine-integration-tests
[...]
PASS:k3s-integration-tests
[...]
PASS:user-accounts-integration-tests
[...]
STOP: ptest-runner
```

Note: `ptest-runner -l` is a useful command to list the available test suites in the image.

Alternatively, a single standalone test suite may be run via a runner script included in the test suite directory:

```
/usr/share/[test-suite-id]/run-[test-suite-id]
```

Upon completion of the test-suite, a result indicator will be output by the script, as one of two options: `PASS:[test-suite-id]` or `FAIL:[test-suite-id]`, as well as an appropriate exit status.

A test suite consists of one or more 'top-level' BATS tests, which may be composed of multiple assertions, where each assertion is considered a named sub-test. If a sub-test fails, its individual result will be included in the output with a

similar format. In addition, if a test failed then debugging information will be provided in the output of type `DEBUG`. The format of these results are described in *Test Logging*.

Test Logging

Test suite execution outputs results and debugging information into a log file. As the test suites are executed using the `test` user account, this log file will be owned by the `test` user and located in the `test` user's home directory by default, at:

```
/home/test/runtime-integration-tests-logs/[test-suite-id].log
```

Therefore, reading this file as another user will require `sudo` access. The location of the log file for each test suite is customizable, as described in the detailed documentation for each test suite below. The log file is replaced on each new execution of a test suite.

The log file will record the results of each top-level integration test, as well as a result for each individual sub-test up until a failing sub-test is encountered.

Each top-level result is formatted as:

```
TIMESTAMP RESULT:[top_level_test_name]
```

Each sub-test result is formatted as:

```
TIMESTAMP RESULT:[top_level_test_name]:[sub_test_name]
```

Where `TIMESTAMP` is of the format `%Y-%m-%d %H:%M:%S` (see Python Datetime Format Codes), and `RESULT` is either `PASS`, `FAIL`, or `SKIP`.

On a test failure, a debugging message of type `DEBUG` will be written to the log. The format of a debugging message is:

```
TIMESTAMP DEBUG:[top_level_test_name]:[return_code]:[stdout]:[stderr]
```

Additional informational messages may appear in the log file with `INFO` or `DEBUG` message types, e.g. to log that an environment clean-up action occurred.

Test Suites

The test suites are detailed below.

Container Engine Tests

The container engine test suite is identified as:

```
container-engine-integration-tests
```

for execution via `ptest-runner` or as a standalone BATS suite, as described in *Running the Tests*.

The test suite is built and installed in the image according to the following BitBake recipe: `meta-cassini-tests/recipes-tests/runtime-integration-tests/container-engine-integration-tests.bb`.

Currently the test suite contains three top-level integration tests, which run consecutively in the following order.

1. `run container` is composed of four sub-tests:
    1.1. Run a containerized detached workload via the `docker run` command
        - Pull an image from the network
        - Create and start a container

1.2. Check the container is running via the `docker inspect` command
1.3. Remove the running container via the `docker remove` command
- Stop the container
- Remove the container from the container list
1.4. Check the container is not found via the `docker inspect` command

2. `container network connectivity` is composed of a single sub-test:
2.1. Run a containerized, immediate (non-detached) network-based workload via the `docker run` command
- Create and start a container, re-using the existing image
- Update package lists within container from external network

The tests can be customized via environment variables passed to the execution, each prefixed by `CE_` to identify the variable as associated to the container engine tests:

`CE_TEST_IMAGE`: defines the container image
Default: `docker.io/library/alpine`
`CE_TEST_LOG_DIR`: defines the location of the log file
Default: `/home/test/runtime-integration-tests-logs/`
Directory will be created if it does not exist
See *Test Logging*
`CE_TEST_CLEAN_ENV`: enable test environment clean-up
Default: `1` (enabled)
See *Container Engine Environment Clean-Up*

Container Engine Environment Clean-Up

A clean environment is expected when running the container engine tests. For example, if the target image already exists within the container engine environment, then the functionality to pull the image over the network will not be validated. Or, if there are running containers from previous (failed) tests then they may interfere with subsequent test executions.

Therefore, if `CE_TEST_CLEAN_ENV` is set to `1` (as is default), running the test suite will perform an environment clean before and after the suite execution.

The environment clean operation involves:

- Determination and removal of all running containers of the image given by `CE_TEST_IMAGE`
- Removal of the image given by `CE_TEST_IMAGE`, if it exists

If enabled then the environment clean operations will always be run, regardless of test-suite success or failure.

K3s Orchestration Tests

The K3s test suite is identified as:

```
k3s-integration-tests
```

for execution via `ptest-runner` or as a standalone BATS suite, as described in *Running the Tests*.

The test suite is built and installed in the image according to the following BitBake recipe within `meta-cassini-tests/recipes-tests/runtime-integration-tests/k3s-integration-tests.bb`.

Currently the test suite contains a single top-level integration test which validates the deployment and high-availability of a test workload based on the Nginx webserver.

The K3s integration tests consider a single-node cluster, which runs a K3s server together with its built-in worker agent. The containerized test workload is therefore deployed to this node for scheduling and execution.

The test suite will not be run until the appropriate K3s services are in the 'active' state, and all 'kube-system' pods are either running, or have completed their workload.

1. `K3s container orchestration` is composed of many sub-tests, grouped here by test area:
   Workload Deployment:
   1.1. Deploy test Nginx workload from YAML file via `kubectl apply`
   1.2. Ensure Pods are initialized via `kubectl wait`
   1.3. Create NodePort Service to expose Deployment via `kubectl create service`
   1.4. Get the IP of the node(s) running the Deployment via `kubectl get`
   1.5. Ensure web service is accessible on the node(s) via `wget`
   Deployment Upgrade:
   1.6. Check initial image version of running Deployment via `kubectl get`
   1.7. Get all pre-upgrade Pod names running Deployment via `kubectl get`
   1.8. Upgrade image version of Deployment via `kubectl set`
   1.9. Ensure a new set of Pod names have been started via `kubectl wait` and `kubectl get`
   1.10. Check Pods are running the upgraded image version via `kubectl get`
   1.11. Ensure web service is still accessible on the node(s) via `wget`
   Server Failure Tolerance:
   1.12. Stop K3s server Systemd service via `systemctl stop`
   1.13. Ensure web service is still accessible on the node(s) via `wget`
   1.14. Restart the Systemd service via `systemctl start`
   1.15. Check K3S server is again responding to `kubectl get`

The tests can be customized via environment variables passed to the execution, each prefixed by `K3S_` to identify the variable as associated to the K3s orchestration tests:

`K3S_TEST_LOG_DIR`: defines the location of the log file
   Default: `/home/test/runtime-integration-tests-logs/`
   Directory will be created if it does not exist
   See *Test Logging*
`K3S_TEST_CLEAN_ENV`: enable test environment clean-up
   Default: `1` (enabled)
   See *K3s Environment Clean-Up*

K3s Environment Clean-Up

A clean environment is expected when running the K3s integration tests, to ensure that the system is ready to be validated. For example, the test suite expects that the Pods created from any previous execution of the integration tests have been deleted, in order to test that a new Deployment successfully initializes new Pods for orchestration.

Therefore, if `K3S_TEST_CLEAN_ENV` is set to 1 (as is default), running the test suite will perform an environment clean before and after the suite execution.

The environment clean operation involves:

- Deleting any previous K3s test Service
- Deleting any previous K3s test Deployment, ensuring corresponding Pods are also deleted

If enabled then the environment clean operations will always be run, regardless of test-suite success or failure.

User Accounts Tests

The User Accounts test suite is identified as:

```
user-accounts-integration-tests
```

for execution via `ptest-runner` or as a standalone BATS suite, as described in *Running the Tests*.

The test suite is built and installed in the image according to the following BitBake recipe within `meta-cassini-tests/recipes-tests/runtime-integration-tests/user-accounts-integration-tests.bb`.

The test suite validates that the user accounts described in *User Accounts* are correctly configured with appropriate access permissions on the Cassini distribution image. The validation performed by the test suite is dependent whether or not it has been configured with *Cassini Security Hardening*.

As the configuration of user accounts is modified for Cassini distribution image which is built with Cassini security hardening, additional security-related validation is included in the test suite for this image. These additional tests validate that the appropriate password requirements and that the mask configuration for permission control of newly created files and directories is applied correctly.

The test suite therefore contains following integration tests:

1. `user accounts management tests` is composed of three sub-tests:
    1.1. Check home directory permissions are correct for the default non-privileged Cassini user account, via the filesystem `stat` utility
    1.2. Check the default privileged Cassini user account has `sudo` command access
    1.3. Check the default non-privileged Cassini user account does not have `sudo` command access
2. `user accounts management additional security tests` is only included for images configured with Cassini security hardening, and is composed of four sub-tests:
    2.1. Log-in to a local console using the non-privileged Cassini user account
        - As part of the log-in procedure, validate the user is prompted to set an account password
    2.2. Check that the umask value is set correctly

The tests can be customized via environment variables passed to the execution, each prefixed by `UA_` to identify the variable as associated to the user accounts tests:

`UA_TEST_LOG_DIR`: defines the location of the log file

Default: `/home/test/runtime-integration-tests-logs/`

Directory will be created if it does not exist

See *Test Logging*

`UA_TEST_CLEAN_ENV`: enable test environment clean-up

Default: `1` (enabled)

See *User Accounts Environment Clean-Up*

User Accounts Environment Clean-Up

As the user accounts integration tests only modify the system for images built with Cassini security hardening, clean-up operations are only performed when running the test suite on these images.

In addition, the clean-up operations will only occur if `UA_TEST_CLEAN_ENV` is set to `1` (as is default).

The environment clean-up operations for images built with Cassini security hardening are:

- Reset the password for the `test` user account
- Reset the password for the non-privileged Cassini user account

After the environment clean-up, the user accounts will return to their original state where the first log-in will prompt the user for a new account password.

If enabled then the environment clean operations will always be run, regardless of test-suite success or failure.

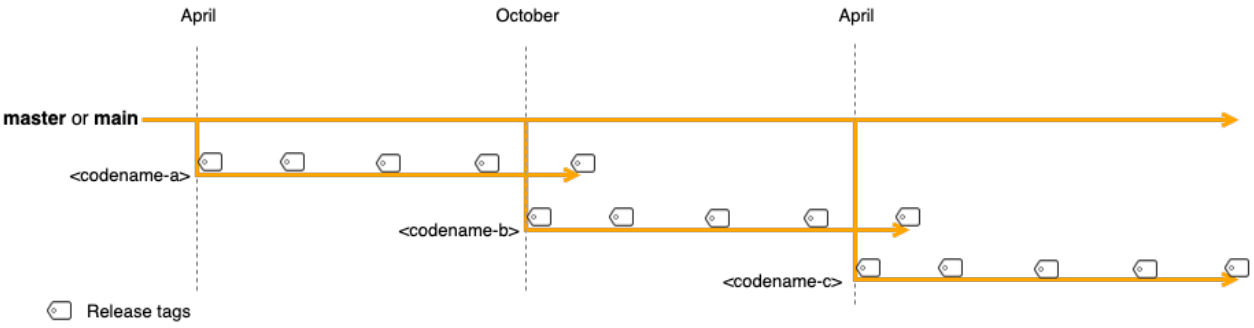
CHAPTER

FOUR

CODELINE MANAGEMENT

The Cassini project is developed and released based on Yocto's release branch process. This strategy allows us to make Major, Minor and Point/Patch Releases based on upstream stable branches, reducing the risk of having build and runtime issues.

4.1 Yocto Release Process Overview

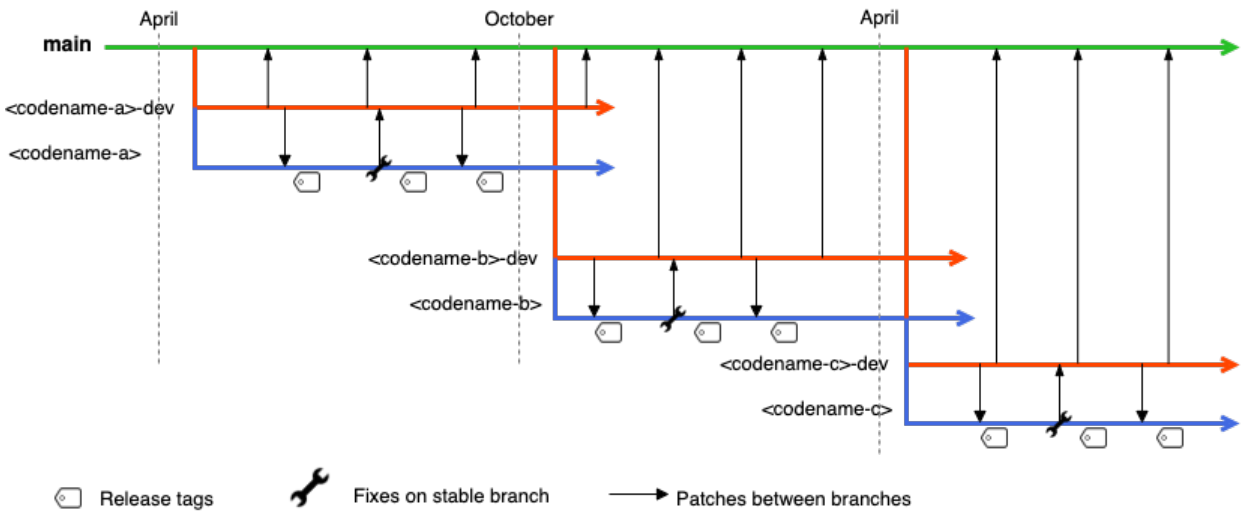


The diagram above gives an overview of the Yocto branch and release process:

- Development happens primarily in the main (or master) branch.
- The project has a major release roughly every 6 months where a stable release branch is created.
- Each major release has a *codename* which is also used to name the stable release branch (e.g. kirkstone).
- Once a stable branch is created and released, it only receives bug fixes with minor (point) releases on an unscheduled basis.
- The goal is for users and 3rd parties layers to use these codenamed branches as a means to be compatible with each other.

For a complete description of the Yocto release process, support schedule and other details, see the Yocto Release Process documentation.

4.2 Cassini Branch and Release Process



Cassini's branch and release process is based on the Yocto release process. The following sub-sections describe in more details the branch strategy for Cassini's development and release process.

4.2.1 Cassini main branch

- Represented by the green line on the diagram above.
- The repository's `main` branch is meant to be compatible with `master` or `main` branches from Poky and 3rd party layers.
- `meta-cassini` is not actively developed on this `main` branch to avoid the instability inherited from Yocto development on the master branch.
- To reduce the effort required to move Cassini to a new version of Yocto, this main branch is periodically updated with patches from the *Cassini development branches* .

4.2.2 Cassini development branches

- Represented by the red line on the diagram above.
- Cassini uses development branches based/compatible with Yocto stable branches.
- A development branch in Cassini is setup for each new Yocto release using the name convention `<codename>-dev` where *<codename>* comes from target Yocto release.
- The development branches in Cassini are where fixes, improvements and new features are developed.
- On a regular basis, code from the development branch is ported over to the `main` branch to reduce the effort required to move Cassini to a new version of Yocto.

4.2.3 Cassini release branches

- Represented by the blue line on the diagram above.
- A new release branch in Cassini is setup for each new Yocto release using the Yocto *codename* the branch targets.
- Hot fixes in the release branch are back ported to the development branch.
- Release branches are currently maintained not much longer than a Yocto release period (~7 months).

4.2.4 Cassini release tags

- Cassini is tagged using the version format `v<Major>.<Minor>.<Patch>`.
- Tags are always applied to commits from the release branch.
- The first release in a release branch is a *Major* release.
- Following releases in a release branch advance the *Minor* version number.
- *Patch* releases are mainly used for hot fixes which are then back ported to the development branch.
- Both *Major* and *Minor* releases may receive fixes, improvements and new features while *Patch* releases only receive fixes. Poky and 3rd party layers release/stable branches might be updated and pinned.

CHAPTER

FIVE

CONTRIBUTING

We welcome contribution from everyone via the `meta-cassini` public Gitlab repository: https://gitlab.arm.com/cassini/meta-cassini. For general introduction about Cassini distribution, refer to *Introduction*.

5.1 License

The Cassini distribution is released under *License*.

Please use an SPDX license identifier in every source file following the recommendations to make it easier for users to understand and review licenses.

```
/* Copyright (c) 2022 Arm Limited or its affiliates. All rights reserved.
* SPDX-License-Identifier: MIT
*/
```

5.2 Contributing to Cassini distribution

This project uses the GitLab project forking workflow. By default, accounts on gitlab.arm.com do not have rights to create personal repositories and therefore fork existing projects. In order to contribute, users must first request access as described here.

Every commit must have at least one `Signed-off-by:` line from the committer to certify that the contribution is made under the terms of the `Developer's Certificate of Origin`.

The full text of `Developer's Certificate of Origin` can be found in sign-your-work-the-developer-s-certificate-of-origin. Due to the significance of the `Developer's Certificate of Origin`, part of it is copied below.

```
The sign-off is a simple line at the end of the explanation for the
patch, which certifies that you wrote it or otherwise have the right to
pass it on as an open-source patch.  The rules are pretty simple: if you
can certify the below:

Developer's Certificate of Origin 1.1
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

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

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

(continues on next page)

(continued from previous page)

```
        indicated in the file; or

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

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

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

then you just add a line saying::

    Signed-off-by: Random J Developer <random@developer.example.org>

using your real name (sorry, no pseudonyms or anonymous contributions.)
```

5.3 Commit guidelines

Commits and patches added should follow the OpenEmbedded patch guidelines.

The component being changed in the shortlog should be prefixed with the layer name (without meta-), for example:

> cassini-config: Decrease frobbing level
>
> cassini-distro: Enable foobar v2
>
> cassini-doc: Added foobar v2 documentation

5.4 Submitting changes

Thank you for your interest in contributing to Cassini distribution. To contribute, follow the *instructions* and ensure you adhere to *commit guidelines*.

5.5 Merge criteria

- The merge request must receive at least 2 approvals from *Cassini distro maintainers*
- `meta-cassini` pipelines are passed
- No regression on code coverage

CHAPTER

SIX

LICENSE

The software is provided under the MIT license (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.
```

6.1 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/