RAUC Documentation

Release 1.5.1

Jan Luebbe, Enrico Joerns, Juergen Borleis

Jan 22, 2021

Contents

Contents:

CHAPTER 1

Updating your Embedded Device

This chapter does not explicitly tell you anything about RAUC itself, but it provides an initial overview of basic requirements and design consideration that have to be taken into account when designing an update architecture for your embedded device.

Thus, if you know about updating and are interested in RAUC itself, only, simply skip this chapter.

Nevertheless, this chapter could also provide some useful hints that can already be useful when designing the device you intend to update later on. In this you initial phase you can prevent yourself from making wrong decisions.

1.1 Redundancy and Atomicity

There are two key requirements for allowing you to robustly update your system.

The first one is redundancy: You must not update the system you are currently running on. Otherwise a failure during updating will brick the only system you can run your update from.

The second one is atomicity: Writing your update to the currently inactive device is a critical operation. A failure occurring during this installation must not brick your device. Thus you must make sure to tell your boot logic to select the updated device not before being very sure that the update successfully completed. Additionally, the operation that switches the boot device must be atomic itself.

1.2 Storage Type and Size

The type and amount of available storage on your device has a huge impact on the design of your updatable embedded system.

Except when optimizing for the smallest storage requirements possible, your system should have two redundant devices or partitions for your root file-system. This full symmetric setup allows you to run your application while safely updating the inactive copy. Additionally, if the running system become corrupted for any reason, you may fall back to you second rootfs device.

If the available storage is not much larger than the space required by your devices rootfs, a full redundant symmetric A/B setup will not be an option. In this case, you might need to use a rescue system consisting of a minimal kernel with an appended initramfs to install your updates.

Note: If you can choose the storage technology for your system, *DO NOT* choose raw NAND flash. NAND (especially MLC) is complex to handle correctly and comes with a variety of very specific effects that may cause difficult to debug problem later (if not all details of the storage stack are configured just right). Instead choose eMMC or SSDs, where the engineers who (hopefully) know the quirks of their technology have created layers that hide this complexity to you.

If storage size can be freely chosen, calculate for at least 2x the size of your rootfs plus additionally required space, e.g. for bootloader, (redundant) data storage, etc.

1.3 Security

An update tool or the infrastructure around it should ensure that no unauthorized entity is able to update your device. This can be done by having:

a) a secure channel to transfer the update or

b) a signed update that allows you to verify its author.

Note that the latter method is more flexible and might be the only option if you intend to use a USB stick for example.

1.4 Interfacing with your Bootloader

The bootloader is the final instance that controls which partition on your rootfs device will be booted. In order to switch partitions after an update, you have to have an interface to the bootloader that allows you to set the boot order, boot priority and other possible parameters.

Some bootloaders, such as U-Boot, allow access to their environment storage where you can freely create and modify variables the bootloader may read. Boot logic often can be implemented by a simple boot script.

Some others have distinct redundancy boot interfaces with redundant state storage. These often provide more features than simply switching boot partitions and are less prone to errors when used. The Barebox bootloader with its bootchooser framework is a good example for this.

1.5 Update Source and Provisioning

Depending on your infrastructure or requirements, an update might be deployed in several ways.

The two most common ones are over network, e.g. by using a deployment server, or simply over a USB stick that will be plugged into the target system.

CHAPTER 2

RAUC Basics

From a top view, the RAUC update framework provides a solution for four basic tasks:

- generating update artifacts
- signing and verification of update artifacts
- robust installation handling
- interfacing with the boot process

RAUC is basically an image-based updater, i.e. it installs file images on devices or partitions. But, for target devices that can have a file system, it also supports installing contents from tar archives. This often provides much more flexibility as a tar does not have to fit a specific partition size or type. RAUC ensures that the target file system will be set up correctly before unpacking the archive.

2.1 Update Artifacts – Bundles

In order to know how to pack multiple file system images, properly handle installation, being able to check system compatibility and for other meta-information RAUC uses a well-defined update artifact format, simply referred to as *bundles* in the following.

A RAUC bundle consists of the file system image(s) or archive(s) to be installed on the system, a *manifest* that lists the images to install and contains options and meta-information, and possible scripts to run before, during or after installation. A bundle may also contain files not referenced in the manifest, such as scripts or archives that are referenced by files that *are* included in the manifest.

To pack this all together, these contents are collected into a SquashFS image. This provides good compression while allowing to mount the bundle without having to unpack it on the target system. This way, no additional intermediate storage is required. For more details see the *Bundle Formats* section.

A key design decision of RAUC is that signing a bundle is mandatory. For development purpose a self-signed certificate might be sufficient, for production the signing process should be integrated with your PKI infrastructure.

Important: A RAUC Bundle should always unambiguously describe the intended target state of the entire system.

2.2 RAUC's System View

Apart from bundle signing and verification, the main task of RAUC is to ensure that all images in your update bundle are copied in the proper way to the proper target device / partition on your board.

In order to allow RAUC to handle your device right, we need to give it the right view on your system.

2.3 Slots

In RAUC, everything that can be updated is a *slot*. Thus a slot can either be a full device, a partition, a volume or simply a file.

To let RAUC know which slots exists on the board that should be handled, the slots must be configured in a *system configuration file*. This file is the central instance that tells RAUC how to handle the board, which bootloader to use, which custom scripts to execute, etc.

The slot description names, for example, the file path the slot can be accessed with, the type of storage or filesystem to use, its identification from the bootloader, etc.

2.4 Target Slot Selection

A very important step when installing an update is to determine the correct mapping from the images that are contained in a RAUC bundle to the slots that are defined on the target system. The updated must also assure to select an inactive slot, and not accidentally a slot the system currently runs from.

For this mapping, RAUC allows to define different *slot classes*. A class describes always multiple redundant slots of the same type. This can be, for example, a class for root file system slots or a class for application slots.

Note that despite the fact that classic A+B redundancy is a common setup for many systems, RAUC conceptually allows any number of redundant slots per class.

Now, multiple slots of different classes can be grouped as a *slot group*. Such a group is the base for the slot selection algorithm of RAUC.

Consider, for example, a system with two redundant rootfs slots and two redundant application slots. Then you group them together to have a fixed set of a rootfs and application slot each that will be used together.

To detect the active slots, RAUC attempts to detect the currently booted slot. For this, it relies on explicit mapping information provided via kernel command line or attempts to find it out using mount information.

All slots of the group containing the active slot will be considered active, too.

2.5 Slot Status and Skipping Slot Updates

RAUC hashes each image or archive when packing it into a bundle and stores this hash in the bundle's manifest file. This hash allows to reliably identify and distinguish the image's content.

When installing an image, RAUC can write the images hash together with some status information to a central or per-slot status file (refer *statusfile* option).

The next time RAUC attempts to install an image to this slot, it will first check the current hash of the slot by reading its status information, if available. If this hash equals the hash of the image to write, RAUC can skip updating this slot as a configurable performance optimization (refer *install-same* per-slot option).

This is especially useful when having a setup with, for example, two redundant application file systems and two redundant root file systems. In case you update the application file system content much more frequently while keeping the exact same rootfs content, RAUC will save update time by skipping the root file system automatically and only installing the changed application.

2.6 Boot Slot Selection

A system designed to run from redundant slots must always have a component that is responsible for selecting between the bootable slots. Usually, this will be some kind of bootloader, but it could also be an initramfs booting a special purpose Linux system.

Of course, as a normal user-space tool, RAUC cannot do the selection itself, but provides a well-defined interface and abstraction for interacting with different bootloaders (e.g. GRUB, Barebox, U-Boot) or boot selection methods.

In order to enable RAUC to switch the correct slot, its system configuration must specify the name of the respective slot from the bootloader's perspective. You also have to set up an appropriate boot selection logic in the bootloader itself, either by scripting (as for GRUB, U-Boot) or by using dedicated boot selection infrastructure (such as bootchooser in Barebox).

The bootloader must also provide a set of variables the Linux userspace can modify in order to change boot order or priority.

Having this interface ready, RAUC will care for setting the boot logic appropriately. It will, for example, deactivate the slot to update before writing to it and reactivate it after having completed the installation successfully.

2.7 Installation and Storage Handling

As mentioned above, RAUC basically writes images to devices or partitions, but also allows installing file system content from (compressed) tar archives.

In addition to the need for different methods to write to storage (simple copy for block devices, nandwrite for NAND, ubiupdatevol for UBI volumes, . . .) the tar-based installation requires additional handling and preparation of storage.

Thus, the possible and required handling depends on both the type of input image (e.g. .tar.xz, .ext4, .img) as well as the type of storage. A tar can be installed on different file systems while an ext4 file system slot might be filled by both an .ext4 image or a tar archive.

To deal with all these possible combinations, RAUC provides an update handler algorithm that uses a matching table to define valid combinations of image and slot type while specifying the appropriate handling.

2.8 Boot Confirmation & Fallback

When designing a robust redundant system, update handling does not end with the successful installation of the update on the target slots! Having written your image data without any errors does not mean that the system you just installed

will really boot. And even if it boots, there may be crashes or invalid behavior only revealed at runtime or possibly not before a number of days and reboots.

To allow the boot logic to detect if booting a slot succeeded or failed, it needs to receive some feedback from the booted system. For marking a boot as either successful or bad, RAUC provides the commands *status mark-good* and *status mark-bad*. These commands interact through the boot loader interface with the respective bootloader implementation to indicate a successful or failed boot.

As detecting an invalid boot is often not possible, i.e. because simply nothing boots or the booted system suddenly crashes, your system should use a hardware watchdog to during boot and have support in the bootloader to detect watchdog resets as failed boots.

Also you need to define what happens when a boot slot is detected to be unusable. For most cases it might be desired to either select one of the redundant slots as fallback or boot into a recovery system. This handling is up to your bootloader.

CHAPTER 3

Using RAUC

For using RAUC in your embedded project, you will need to build at least two versions of it:

- One for your **host** (build or development) system. This will allow you to create, inspect and modify bundles.
- One for your **target** system. This can act both as the service for handling the installation on your system, as a command line tool that allows triggering the installation and inspecting your system or obtaining bundle information.

All common embedded Linux build system recipes for RAUC will solve the task of creating appropriate binaries for you as well as caring for bundle creation and partly system configuration. If you intend to use RAUC with Yocto, use the meta-rauc layer, in case you use PTXdist, simply enable RAUC in your configuration.

Note: When using the RAUC service from your application, the D-Bus interface is preferable to using the provided command-line tool.

3.1 Creating Bundles

To create an update bundle on your build host, RAUC provides the `bundle` sub-command:

```
rauc bundle --cert=<certfile> --key=<keyfile> --keyring=<keyringfile> <input-dir>
↪<output-file>
```

Where `<input-dir>` must be a directory containing all images and scripts the bundle should include, as well as a manifest file `manifest.raucm` that describes the content of the bundle for the RAUC updater on the target: which image to install to which slot, which scripts to execute etc. Note that all files in `<input-dir>` will be included in the bundle, not just those specified in the manifest (see also the *example* and the *reference*). `<output-file>` must be the path of the bundle file to create.

Instead of the `certfile` and `keyfile` arguments, PKCS#11 URLs such as `'pkcs11:token=rauc;object=autobuilder-1'` can be used to avoid storing sensitive key material as files (see *PKCS#11 Support* for details).

While the `--cert` and `--key` argument are mandatory for signing and must provide the certificate and private key that should be used for creating the signature, the `--keyring` argument is optional and (if given) will be used for verifying the trust chain validity of the signature after creation. Note that this is very useful to prevent signing with obsolete certificates, etc.

3.2 Obtaining Bundle Information

```
rauc info [--output-format=<format>] <input-file>
```

The `info` command lists the basic meta data of a bundle (compatible, version, build-id, description) and the images and hooks contained in the bundle.

You can control the output format depending on your needs. By default it will print a human readable representation of the bundle not intended for being processed programmatically. Alternatively you can obtain a shell-parsable description or a JSON representation of the bundle content.

3.3 Installing Bundles

To actually install an update bundle on your target hardware, RAUC provides the `install` command:

```
rauc install <input-file>
```

Alternatively you can trigger a bundle installation *using the D-Bus API*.

3.4 Viewing the System Status

For debugging purposes and for scripting it is helpful to gain an overview of the current system as RAUC sees it. The `status` command allows this:

```
rauc status [--detailed] [--output-format=<format>]
```

You can choose the output style of RAUC status depending on your needs. By default it will print a human readable representation of your system's most important properties. Alternatively you can obtain a shell-parsable description, or a JSON representation of the system status. If more information is needed such as the slots' *status* add the command line option `--detailed`.

3.5 React to a Successfully Booted System/Failed Boot

Normally, the full system update chain is not complete before being sure that the newly installed system runs without any errors. As the definition and detection of a *successful* operation is really system-dependent, RAUC provides commands to preserve a slot as being the preferred one to boot or to discard a slot from being bootable.

```
rauc status mark-good
```

After verifying that the currently booted system is fully operational, one wants to signal this information to the underlying bootloader implementation which then, for example, resets a boot attempt counter.

```
rauc status mark-bad
```

If the current boot failed in some kind, this command can be used to communicate that to the underlying bootloader implementation. In most cases this will disable the currently booted slot or at least switch to a different one.

Although not very useful in the field, both commands recognize an optional argument to explicitly identify the slot to act on:

```
rauc status mark-{good,bad} [booted | other | <SLOT_NAME>]
```

This is to maintain consistency with respect to `rauc status mark-active` where that argument is definitively wanted, see *here*.

3.6 Manually Switch to a Different Slot

One can think of a variety of reasons to switch the preferred slot for the next boot by hand, for example:

- Recurrently test the installation of a bundle in development starting from a known state.
- Activate a slot that has been installed sometime before and whose activation has explicitly been prevented at that time using the system configuration file's parameter *activate-installed*.
- Switch back to the previous slot because one really knows better™.

To do so, RAUC offers the subcommand

```
rauc status mark-active [booted | other | <SLOT_NAME>]
```

where the optional argument decides which slot to (re-)activate at the expense of the remaining slots. Choosing `other` switches to the next bootable slot that is not the one that is currently booted. In a two-slot-setup this is just... the other one. If one wants to explicitly address a known slot, one can do so by using its slot name which has the form `<slot-class>.<idx>` (e.g. `rootfs.1`), see *this* part of section *System Configuration File*. Last but not least, after switching to a different slot by mistake, before having rebooted this can be remedied by choosing `booted` as the argument which is, by the way, the default if the optional argument has been omitted. The date and time of activation as well as the number of activations is part of the slot's metadata which is stored in the slot status file, see section *Slot Status*.

3.7 Customizing the Update

RAUC provides several ways to customize the update process. Some allow adding and extending details more fine-grainedly, some allow replacing major parts of the default behavior of RAUC.

In general, there exist three major types of customization:

- configuration parameters (in rootfs config file `/etc/rauc/system.conf`)
- handlers (executables in rootfs)
- hooks (executables in bundle)

The first type, configuration parameters, allow controlling parameters of the update in a predefined way.

The second type, using *handlers*, allows extending or replacing the installation process. They are executables (most likely shell scripts) located in the root filesystem and configured in the system's configuration file. They control static behavior of the system that should remain the same over future updates.

The last type are *hooks*. They are similar to *handlers*, except that they are contained in the update bundle. Thus they allow to flexibly extend or customize one or more updates by some special behavior. A common example would be

using a per-slot post-install hook that handles configuration migration for a new software version. Hooks are especially useful to handle details of installing an update which were not considered in the previously deployed version.

In the following, configuration parameters, handlers and hooks will be explained in more detail.

3.7.1 System Configuration Parameters

Beside providing the basic slot layout, RAUC's system configuration file (`system.conf`) also allows you to configure parts of its runtime behavior, such as handlers (see below), paths, etc. For a detailed list of possible configuration options, see *System Configuration File* section in the *Reference* chapter.

3.7.2 System-Based Customization: Handlers

Handlers are executables located in the target's *root file system* that allow extending the installation process on system side. They must be specified in the targets *System Configuration File*.

For a detailed list of all environment variables exported for the handler scripts, see the *Custom Handlers (Interface)* section.

Pre-Install Handler

```
[handlers]
pre-install=/usr/lib/rauc/pre-install
```

RAUC will call the pre-install handler (if given) during the bundle installation process, right before calling the default or custom installation process. At this stage, the bundle is mounted, its content is accessible and the target group has been determined successfully.

If calling the handler fails or the handler returns a non-zero exit code, RAUC will abort installation with an error.

Post-Install Handler

```
[handlers]
post-install=/usr/lib/rauc/post-install
```

The post-install handler will be called right after RAUC successfully performed a system update. If any error occurred during installation, the post-install handler will not be called.

Note that a failed call of the post-install handler or a non-zero exit code will cause a notification about the error but will not change the result of the performed update anymore.

A possible usage for the post-install handler could be to trigger an automatic restart of the system.

System-Info Handler

```
[handlers]
system-info=/usr/lib/rauc/system-info
```

The system-info handler is called after loading the configuration file. This way it can collect additional variables from the system, like the system's serial number.

The handler script must return a system serial number by echoing *RAUC_SYSTEM_SERIAL=<value>* to standard out.

3.7.3 Bundle-Based Customization: Hooks

Unlike handlers, hooks are part of the update bundle and must be specified in the bundle's *Manifest* file and handled by a common executable. Hooks allow the author of a bundle to add or replace functionality for the installation of a specific bundle. This can be useful for performing additional migration steps, checking for specific previously installed bundle versions or for manually handling updates of images RAUC cannot handle natively.

To reduce the complexity and number of files in a bundle, all hooks must be handled by a single executable that is registered in the bundle's manifest:

```
[hooks]
filename=hook
```

Each hook must be activated explicitly and leads to a call of the hook executable with a specific argument that allows to distinguish between the different hook types. Multiple hook types must be separated with a `;`.

In the following the available hooks are listed. Depending on their purpose, some are image-specific, i.e. they will be executed for the installation of a specific image only, while some other are global.

Install Hooks

Install hooks operate globally on the bundle installation.

For a detailed list of all environment variables exported for the hooks executable, see the *Install Hooks Interface* section.

Install-Check Hook

```
[hooks]
filename=hook
hooks=install-check
```

This hook will be executed instead of the normal compatible check in order to allow performing a custom compatibility check based on compatible and/or version information.

To indicate that a bundle should be rejected, the script must return with an exit code >= 10.

If available, RAUC will use the last line printed to standard error by the hook executable as the rejection reason message and provide it to the user:

```
#!/bin/sh

case "$1" in
        install-check)
                if [[ "$RAUC_MF_COMPATIBLE" != "$RAUC_SYSTEM_COMPATIBLE" ]]; then
                        echo "Compatible does not match!" 1>&2
                        exit 10
                fi
                ;;
        *)
                exit 1
                ;;
esac

exit 0
```

Slot Hooks

Slot hooks are called for each slot an image will be installed to. In order to enable them, you have to specify them in the `hooks` key under the respective `image` section.

Note that hook slot operations will be passed to the executable with the prefix `slot-`. Thus if you intend to check for the pre-install hook, you have to check for the argument to be `slot-pre-install`.

For a detailed list of all environment variables exported for the hooks executable, see the *Slot Hooks Interface* section.

Pre-Install Hook

The pre-install hook will be called right before the update procedure for the respective slot will be started. For slot types that represent a mountable file system, the hook will be executed with having the file system mounted.

```
[hooks]
filename=hook

[image.rootfs]
filename=rootfs.img
size=...
sha256=...
hooks=pre-install
```

Post-Install Hook

The post-install hook will be called right after the update procedure for the respective slot was finished successfully. For slot types that represent a mountable file system, the hook will be executed with having the file system mounted. This allows to write some post-install information to the slot. It is also useful to copy files from the currently active system to the newly installed slot, for example to preserve application configuration data.

```
[hooks]
filename=hook

[image.rootfs]
filename=rootfs.img
size=...
sha256=...
hooks=post-install
```

An example on how to use a post-install hook:

```
#!/bin/sh

case "$1" in
        slot-post-install)
                # only rootfs needs to be handled
                test "$RAUC_SLOT_CLASS" = "rootfs" || exit 0

                touch "$RAUC_SLOT_MOUNT_POINT/extra-file"
                ;;
        *)
                exit 1
                ;;
esac
```

(continues on next page)

(continued from previous page)

```
exit 0
```

Install Hook

The install hook will replace the entire default installation process for the target slot of the image it was specified for. Note that when having the install hook enabled, pre- and post-install hooks will *not* be executed. The install hook allows to fully customize the way an image is installed. This allows performing special installation methods that are not natively supported by RAUC, for example to upgrade the bootloader to a new version while also migrating configuration settings.

```
[hooks]
filename=hook

[image.rootfs]
filename=rootfs.img
size=...
sha256=...
hooks=install
```

3.7.4 Full Custom Update

For some special tasks (recovery, testing, migration) it might be required to completely replace the default RAUC update mechanism and to only use its infrastructure for executing an application or a script on the target side.

For this case, you may replace the entire default installation handler of rauc by a custom handler script or application.

Refer manifest *[handler]* section description on how to achieve this.

3.8 Using the D-Bus API

The RAUC D-BUS API allows seamless integration into existing or project-specific applications, incorporation with bridge services such as the *rauc-hawkbit* client and also the rauc CLI uses it.

The API's service domain is `de.pengutronix.rauc` while the object path is `/`.

3.8.1 Installing a Bundle

The D-Bus API's main purpose is to trigger and monitor the installation process via its `Installer` interface.

The `InstallBundle` method call triggers the installation of a given bundle in the background and returns immediately. Upon completion of the installation RAUC emits the `Completed` signal, indicating either successful or failed installation. For details on triggering the installation process, see the *The InstallBundle() Method* chapter in the reference documentation.

While the installation is in progress, constant progress information will be emitted in form of changes to the `Progress` property.

3.8.2 Processing Progress Data

The progress property will be updated upon each change of the progress value. For details see the *The "Progress" Property* chapter in the reference documentation.

To monitor `Progress` property changes from your application, attach to the `PropertiesChanged` signal and filter on the `Operation` properties.

Each progress step emitted is a tuple `(percentage, message, nesting depth)` describing a tree of progress steps:

```
├"Installing" (0%)
│ ├"Determining slot states" (0%)
│ ├"Determining slot states done." (20%)
│ ├"Checking bundle" (20%)
│ │ ├"Verifying signature" (20%)
│ │ └"Verifying signature done." (40%)
│ ├"Checking bundle done." (40%)
│ ...
└"Installing done." (100%)
```

This hierarchical structure allows applications to decide for the appropriate granularity to display information. Progress messages with a nesting depth of 1 are only `Installing` and `Installing done.`. A nesting depth of 2 means more fine-grained information while larger depths are even more detailed.

Additionally, the nesting depth information allows the application to print tree-like views as shown above. The `percentage` value always goes from 0 to 100 while the `message` is always a human-readable English string. For internationalization you may use a gettext-based approach.

3.8.3 Examples Using `busctl` Command

Triggering an installation:

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer InstallBundle sa{sv}
↪"/path/to/bundle" 0
```

Mark a slot as good:

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer Mark ss "good"
↪"rootfs.0"
```

Mark a slot as active:

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer Mark ss "active"
↪"rootfs.0"
```

Get the *Operation* property containing the current operation:

```
busctl get-property de.pengutronix.rauc / de.pengutronix.rauc.Installer Operation
```

Get the *Progress* property containing the progress information:

```
busctl get-property de.pengutronix.rauc / de.pengutronix.rauc.Installer Progress
```

Get the *LastError* property, which contains the last error that occurred during an installation.

```
busctl get-property de.pengutronix.rauc / de.pengutronix.rauc.Installer LastError
```

Get the status of all slots

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer GetSlotStatus
```

Get the current primary slot

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer GetPrimary
```

Monitor the D-Bus interface

```
busctl monitor de.pengutronix.rauc
```

3.9 Debugging RAUC

When RAUC fails to start on your target during integration or later during installation of new bundles it can have a variety of causes.

This section will lead you trough the most common options you have for debugging what actually went wrong.

In each case it is quite essential to know that RAUC, if not compiled with `--disable-service` runs as a service on your target that is either controlled by your custom application or by the RAUC command line interface.

The frontend will always only show the 'high level' error outpt, e.g. when an installation failed:

```
rauc-Message: 08:27:12.083: installing /home/enrico/Code/rauc/good-bundle-hook.raucb:␣
↪LastError: Failed mounting bundle: failed to run mount: Child process exited with␣
↪code 1
rauc-Message: 08:27:12.083: installing /home/enrico/Code/rauc/good-bundle-hook.raucb:␣
↪idle
Installing `/home/enrico/Code/rauc/good-bundle-hook.raucb` failed
```

In simple cases this might be sufficient for identifying the actual problem, in more complicated cases this may give a rough hint. For a more detailed look on what went wrong you need to inspect the rauc service log instead.

If you run RAUC using systemd, the log can be obtained using

```
journalctl -u rauc
```

When using SysVInit, your service script needs to configure logging itself. A common way is to dump the log e.g. /var/log/rauc.

It may also be worth starting the RAUC service via command line on a second shell to have a live view of what is going on when you invoke e.g. `rauc install` on the first shell.

3.9.1 Increasing Debug Verbosity

Both for the service and the command line interface it is often useful to increase the log level for narrowing down the actual error cause or gaining more information about the circumstances when the error occurs.

RAUC uses glib and the glib logging framework with the basic log domain 'rauc'.

For simple cases, you can activate logging by passing the `-d` or `--debug` option to either the CLI:

```
rauc install -d bundle.raucb ..
```

or the service (you might need to modify your systemd or SysVInit service file).

```
rauc service -d
```

For more fine grained and advanced debugging options, use the `G_MESSAGES_DEBUG` environment variable. This allows enabling different log domains. Currently available are:

all enable all log domains

rauc enable default RAUC log domain (same as calling with `-d`)

rauc-signature enable logging of signature details

This will dump the full CMS structure during verification and can help identify problems with the signature details.

rauc-subprocess enable logging of subprocess calls

This will dump the entire program call invoked by RAUC and can help tracing down or reproducing issues caused by other programs invoked.

Example invocation:

```
G_MESSAGES_DEBUG="rauc rauc-subprocess" rauc service
```

CHAPTER 4

Examples

4.1 Full System Example

This chapter aims to explain the basic concepts needed for RAUC using a simple but realistic scenario.

The system is x86-based with 1GiB of disk space and 1GiB of RAM. GRUB was selected as the bootloader and we want to have two symmetric installations. Each installation consists of an ext4 root file system only (which contains the matching kernel image).

We want to provide update bundles using a USB memory stick. We don't have a hardware watchdog, so we need to explicitly tell GRUB whether a boot was successful.

This scenario can be easily reproduced using a QEMU virtual machine.

4.1.1 PKI Setup

RAUC uses an x.509 PKI (public key infrastructure) to sign and verify updates. To create a simple key pair for testing, we can use `openssl`:

```
> openssl req -x509 -newkey rsa:4096 -nodes -keyout demo.key.pem -out demo.cert.pem -
↪subj "/O=rauc Inc./CN=rauc-demo"
```

For actual usage, setting up a real PKI (with a CA separate from the signing keys and a revocation infrastructure) is *strongly* recommended. OpenVPN's easy-rsa is a good first step. See *Security* for more details.

4.1.2 RAUC Configuration

We need a RAUC system configuration file to describe the slots which can be updated

```
[system]
compatible=rauc-demo-x86
bootloader=grub
```

(continues on next page)

(continued from previous page)

```
mountprefix=/mnt/rauc
bundle-formats=-plain

[keyring]
path=demo.cert.pem

[slot.rootfs.0]
device=/dev/sda2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/sda3
type=ext4
bootname=B
```

In this case, we need to place the signing certificate into `/etc/rauc/demo.cert.pem`, so that it is used by RAUC for verification.

4.1.3 GRUB Configuration

GRUB itself is stored on `/dev/sda1`, separate from the root file system. To access GRUB's environment file, this partition should be mounted to `/boot` (which means that the environment file is found at `/boot/grub/grubenv`).

GRUB does not provide the boot target selection logic as needed by RAUC out of the box. Instead we use a script to implement it

```
default=0
timeout=3

set ORDER="A B"
set A_OK=0
set B_OK=0
set A_TRY=0
set B_TRY=0
load_env

# select bootable slot
for SLOT in $ORDER; do
    if [ "$SLOT" == "A" ]; then
        INDEX=0
        OK=$A_OK
        TRY=$A_TRY
        A_TRY=1
    fi
    if [ "$SLOT" == "B" ]; then
        INDEX=1
        OK=$B_OK
        TRY=$B_TRY
        B_TRY=1
    fi
    if [ "$OK" -eq 1 -a "$TRY" -eq 0 ]; then
        default=$INDEX
        break
    fi
```

(continues on next page)

(continued from previous page)

```
done

# reset booted flags
if [ "$default" -eq 0 ]; then
    if [ "$A_OK" -eq 1 -a "$A_TRY" -eq 1 ]; then
        A_TRY=0
    fi
    if [ "$B_OK" -eq 1 -a "$B_TRY" -eq 1 ]; then
        B_TRY=0
    fi
fi

save_env A_TRY B_TRY

CMDLINE="panic=60 quiet"

menuentry "Slot A (OK=$A_OK TRY=$A_TRY)" {
    linux (hd0,2)/kernel root=/dev/sda2 $CMDLINE rauc.slot=A
}

menuentry "Slot B (OK=$B_OK TRY=$B_TRY)" {
    linux (hd0,3)/kernel root=/dev/sda3 $CMDLINE rauc.slot=B
}
```

GRUB since 2.02-beta1 supports the `eval` command, which can be used to express the logic above more concisely.

The `grubenv` file can be modified using `grub-editenv`, which is shipped by GRUB. It can also be used to inspect the current contents:

```
> grub-editenv /boot/grub/grubenv list
ORDER="A B"
A_OK=0
B_OK=0
A_TRY=0
B_TRY=0
```

The initial installation of the bootloader and rootfs on the system is out of scope for RAUC. A common approach is to generate a complete disk image (including the partition table) using a build system such as OpenEmbedded/Yocto, PTXdist or buildroot.

4.1.4 Bundle Generation

To create a bundle, we need to collect the components which should become part of the update in a directory (in this case only the root file system image):

```
> mkdir temp-dir/
> cp .../rootfs.ext4.img temp-dir/
```

Next, to describe the bundle contents to RAUC, we create a *manifest* file. This must be named `manifest.raucm`:

```
> cat >> temp-dir/manifest.raucm << EOF
[update]
compatible=rauc-demo-x86
version=2015.04-1

```

(continues on next page)

(continued from previous page)

```
[bundle]
format=verity

[image.rootfs]
filename=rootfs.ext4.img
EOF
```

Note that we can omit the `sha256` and `size` parameters for the image here, as RAUC will fill them out automatically when creating the bundle.

Finally, we run RAUC to create the bundle:

```
> rauc --cert demo.cert.pem --key demo.key.pem bundle temp-dir/ update-2015.04-1.raucb
> rm -r temp-dir
```

We now have the `update-2015.04-1.raucb` bundle file, which can be copied onto the target system, in this case using a USB memory stick.

4.1.5 Update Installation

Having copied `update-2015.04-1.raucb` onto the target, we only need to run RAUC:

```
> rauc install /mnt/usb/update-2015.04-1.raucb
```

After cyptographically verifying the bundle, RAUC will now determine the active slots by looking at the `rauc.slot` variable. Then, it can select the target slot for the update image from the inactive slots.

When the update is installed completely, we just need to restart the system. GRUB will then try to boot the newly installed rootfs. Finally, if the boot was successful, we need to inform the bootloader:

```
> rauc status mark-good
```

If systemd is available, it is useful to run this command late in the boot process and declare dependencies on the main application(s).

If the boot is not marked as successful, GRUB will try the other installation on the next boot. By configuring the kernel and systemd to reboot on critical errors and by using a (software) watchdog, hangs in a non-working installation can be avoided.

4.1.6 Write Slots Without Update Mechanics

Assuming an image has been copied to or exists on the target, a manual slot write can be performed by:

```
> rauc write-slot rootfs.0 rootfs.ext4
```

This will write the rootfs image `rootfs.ext4` to the slot `rootfs.0`. Note that this bypasses all update mechanics like hooks, slot status etc.

4.2 Example Slot Configurations

This provides some common examples on how to configure slots in your system.conf for different scenarios.

4.2.1 Symmetric A/B Setup

This is the default case when having a fully-redundant root file system

```
[...]

[slot.rootfs.0]
device=/dev/sda2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/sda3
type=ext4
bootname=B
```

4.2.2 Asymmetric A/Recovery Setup

In case storage is too restricted for a full A/B redundancy setup, an asymmetric setup with a dedicated update/recovery slot can be used. The recovery slot can be way smaller than the rootfs one as it needs to contain only the tools for updating the rootfs slot. Because the recovery slot is not meant to be updated in most cases, we can manifest this for RAUC by setting the `readonly=true` option.

```
[...]

[slot.recovery.0]
device=/dev/sda2
type=ext4
bootname=R
readonly=true

[slot.rootfs.0]
device=/dev/sda3
type=ext4
bootname=A
```

4.2.3 Separate Application Partition

RAUC allows to have a separate redundant set of slots for the application (or other purpose) that have a fixed relation to their corresponding rootfs slots. RAUC assures that an update of the entire slot group (rootfs + appfs) is atomic.

When defining appfs slots, be sure to set the correct *parent* relation to the associated bootable slot.

```
[...]

[slot.rootfs.0]
device=/dev/sda2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/sda3
type=ext4
bootname=B
```

(continues on next page)

(continued from previous page)

```
[slot.appfs.0]
parent=rootfs.0
device=/dev/sda4
type=ext4

[slot.appfs.1]
parent=rootfs.1
device=/dev/sda5
type=ext4
```

4.2.4 Atomic Bootloader Updates (eMMC)

Updating the Bootloader is also possible with RAUC, despite this is a bit more critical than updating the rootfs, as there is no fallback mechanism.

However, depending on the ROM loader it can at least be possible to perform the bootloader update atomically. The most common example for this is using the two boot partitions of an eMMC for atomic bootloader updates which RAUC supports out-of-the-box (refer *Update eMMC Boot Partitions*).

```
[...]

[slot.bootloader.0]
device=/dev/mmcblk0
type=boot-emmc

[slot.rootfs.0]
device=/dev/mmcblk0p1
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/mmcblk0p2
type=ext4
bootname=B
```

4.2.5 Symmetric A/B Setup + Recovery

Booting into the recovery slot should normally be handled by the bootloader if it fails to load the symmetric slots.

Thus from the RAUC perspective this setup is identical to the default A/B setup.

Anyway, you can still define it as a slot if you need to be able to provide an update for this, too.

4.3 Example BSPs

- Yocto
- PTXdist

CHAPTER 5

Scenarios

5.1 Symmetric Root-FS Slots

This is the probably the most common setup. In this case, two root partitions of the same size are used (often called "A" and "B"). When running from "A", an update is installed into "B" and vice versa. Both slots are intended to contain equivalent software, including the main application.

To reduce complexity, the kernel and other files necessary for booting the system (such as the device tree) are stored in the root-fs partition (usually in /boot). This requires a boot-loader with support for the root-fs type.

The RAUC `system.conf` would contain two slots similar to the following:

```
[slot.rootfs.0]
device=/dev/sda0
type=ext4
bootname=system-a

[slot.rootfs.1]
device=/dev/sda1
type=ext4
bootname=system-b
```

The main advantage of this setup is its simplicity:

- An update can be started when running in either slot and while the main application is still active.
- The fallback logic in the boot-loader can be relatively simple.
- Easy to understand update process for end-users and technicians.

The main reasons for not using it are either:

- Too limited storage space (use asymmetric slots instead)
- Additional requirements regarding redundancy or update flexibility (see below)

5.2 Asymmetric Slots

This setup is useful if the storage space is very limited. Instead of requiring two partitions each large enough for the full installation, a small partition is used instead of the second one (often called "main" and "update" or "rescue").

The slot configuration for this in `system.conf` could look like this:

```
[slot.update.0]
device=/dev/sda0
type=raw
bootname=update

[slot.main.1]
device=/dev/sda1
type=ext4
bootname=main
```

To update the main system, a reboot into the update system is needed (as otherwise the main slot would still be active). Then, the update system would trigger the installation into the main slot and finally switch back to the newly updated main system. The update system itself can be updated directly from the running main system.

Some disadvantages of this configuration are:

- Two reboots are required for an update.
- A failed update results in an unavailable main application until a subsequent update is installed successfully.
- If some data in the main slot needs to be preserved during the update, it must be stored somewhere else before writing the new image to the slot and then restored.

As the update system is normally small enough to fit completely into RAM, it can be stored as a Linux kernel with internal initramfs. This avoids compressing kernel and user-space separately, increasing the compression ratio. For this, the update slot type should be configured to `raw`.

5.3 Multiple Slots

Splitting a system into multiple slots can be useful if the application should be updated independently of the base system. This can be combined with either symmetric or asymmetric setups as described above.

For example, the main application could be split of from the root file-system. This can be useful if the base system is developed independently from the application(s) or by a different team. By explicitly distinguishing between the two, different versions of the application or even completely different applications can reuse the same base system (root-file-system).

Another reason to configure multiple slots for one system can be to store the boot files (kernel, ...) separately, which can help reduce boot time and complexity in the boot-loader.

```
[slot.rootfs.0]
device=/dev/sda0
type=ext4
bootname=system-a

[slot.appfs.0]
device=/dev/sda1
type=ext4
parent=rootfs.0
```

(continues on next page)

(continued from previous page)

```
[slot.rootfs.1]
device=/dev/sdb0
type=ext4
bootname=system-b

[slot.appfs.1]
device=/dev/sdb1
type=ext4
parent=rootfs.1
```

Warning: Currently, RAUC has no way to ensure compatibility between rootfs and appfs when installing a bundle containing only an image for one of them. Either always build bundles containing images for all required slots or ensure that incompatible updates are not installed outside of RAUC. To solve this, a bundle would need to contain the metadata (size and hash) for the missing bundle and RAUC would need to verify the state of those slots before installing the bundle.

5.4 Additional Rescue Slot

By adding an additional rescue (or recovery) slot to one of the symmetric scenarios above, the robustness against some error cases can be improved:

- A software error has remained undetected over some releases, rendering both normal slots inoperable over time.
- The normal slots are mounted read-write during normal operation and have become corrupted (for example by incorrect handling of sudden power failures).
- A configuration error causes both normal slots to fail in the same way.

```
[slot.rescue.0]
device=/dev/sda0
type=raw
bootname=rescue

[slot.rootfs.0]
device=/dev/sda1
type=ext4
bootname=system-a

[slot.rootfs.1]
device=/dev/sda2
type=ext4
bootname=system-b
```

The rescue slot would not be changed by normal updates (which only write to A and B in turn). Depending on the use case, the boot-loader would start the rescue system after repeated boot failures of the normal systems or on user request.

CHAPTER 6

Integration

If you intend to prepare your platform for using RAUC as an update framework, this chapter will guide you through the required steps and show the different ways you can choose.

To integrate RAUC, you first need to be able to build RAUC as both a host and a target application. The host application is needed for generating update bundles while the target application or service performs the core task of RAUC: updating you device.

In an update system, a lot of components have to play together and have to be configured appropriately to interact correctly. In principle, these are:

- Hardware setup, devices, partitions, etc.
- The bootloader
- The Linux kernel
- The init system
- System utilities (mount, mkfs, . . .)
- The update tool, RAUC itself

Note: When integrating RAUC into your embedded Linux system, and in general, we highly recommend using a Linux system build system like Yocto / OpenEmbedded or PTXdist that allows you to have well defined software states while easing integration of the different components involved.

For information about how to integrate RAUC using these tools, refer to the sections *Yocto* or *PTXdist*.

6.1 RAUC System Configuration

The system configuration file is the central configuration in RAUC that abstracts the loosely coupled storage setup, partitioning and boot strategy of your board to a coherent redundancy setup world view for RAUC.

RAUC expects its central configuration file `/etc/rauc/system.conf` to describe the system it runs on in a way that all relevant information for performing updates and making decisions are given.

Note: For a full reference of the system.conf file refer to section *System Configuration File*.

Similar to other configuration files used by RAUC, the system configuration uses a key-value syntax (similar to those known from .ini files).

6.1.1 Slot Configuration

The most important step is to describe the slots that RAUC should use when performing updates. Which slots are required and what you have to take care of when designing your system will be covered in the chapter *Scenarios*. This section assumes that you have already decided on a setup and want to describe it for RAUC.

A slot is defined by a slot section. The naming of the section must follow a simple format: `[slot.<slot-class>.<slot-index>]` where <*slot-class*> describes a class of possibly multiple redundant slots (such as `rootfs`, `recovery` or `appfs`) and *slot-index* is the index of the individual slot instance, starting with index 0.

If you have two redundant slots used for the root file system, for example, you should name your sections according to this example:

```
[slot.rootfs.0]
device = [...]

[slot.rootfs.1]
device = [...]
```

RAUC does not have predefined class names. The only requirement is that the class names used in the system config match those you later use in the update manifests.

The mandatory settings for each slot are:

- the `device` that holds the (device) path describing *where* the slot is located,
- the `type` that defines *how* to update the target device.

If the slot is bootable, then you also need

- the `bootname` which is the name the bootloader uses to refer to this slot device.

Slot Type

A list of slot storage types currently supported by RAUC:

Type	Description	Tar support
raw	A partition holding no (known) file system. Only raw image copies may be performed.	
ext4	A block device holding an ext4 filesystem.	x
nand	A raw NAND partition.	
ubivol	An UBI partition in NAND.	
ubifs	An UBI volume containing an UBIFS in NAND.	x
vfat	A block device holding a vfat filesystem..	x

Depending on this slot storage type and the slot's *image filename* extension, RAUC determines how to extract the image content to the target slot.

While the generic filename extension `.img` is supported for all filesystems, it is strongly recommended to use explicit extensions (e.g. `.vfat` or `.ext4`) when possible, as this allows checking during installation that the slot type is correct.

Grouping Slots

If multiple slots belong together in a way that they always have to be updated together with the respective other slots, you can ensure this by grouping slots.

A group must always have a single bootable slot, then all other slots define a parent relationship to this bootable slot as follows:

```
[slot.rootfs.0]
...

[slot.appfs.0]
parent = rootfs.0
...

[slot.rootfs.1]
...

[slot.appfs.1]
parent = rootfs.1
...
```

6.2 Library Dependencies

The minimal requirement for RAUC regardless of whether intended for the host or target side is GLib (minimum version 2.45.8) as utility library and OpenSSL (>=1.0) for signature handling.

Note: In order to let RAUC detect mounts correctly, GLib must be compiled with libmount support (`--enable-libmount`) and at least be 2.49.5.

For network support (enabled with `--enable-network`), additionally *libcurl* is required. This is only useful for the target service.

For JSON-style support (enabled with `--enable-json`), additionally *libjson-glib* is required.

6.3 Kernel Configuration

The kernel used on the target device must support both loop block devices and the SquashFS file system to allow installing RAUC bundles. For the recommended `verity` *bundle format*, dm-verity must be supported as well.

In kernel Kconfig you have to enable the following options:

```
CONFIG_MD=y
CONFIG_BLK_DEV_DM=y
CONFIG_BLK_DEV_LOOP=y
CONFIG_DM_VERITY=y
CONFIG_SQUASHFS=y
```

Note: These drivers may also be loaded as modules. Kernel versions v5.0 to v5.7 will require the patch `7e81f99afd91c937f0e66dc135e26c1c4f78b003` backporting to fix a bug where the bundles cannot be mounted in a small number of cases.

6.4 Required Host Tools

To be able to generate bundles, RAUC requires at least the following host tools:

- mksquashfs
- unsquashfs

When using the RAUC casync integration, the `casync` tool and `fakeroot` (for converting archives to directory tree indexes) must also be available.

6.5 Required Target Tools

RAUC requires and uses a set of target tools depending on the type of supported storage and used image type.

Mandatory tools for each setup are `mount` and `umount`, either from Busybox or util-linux

Note that build systems may handle parts of these dependencies automatically, but also in this case you will have to select some of them manually as RAUC cannot fully know how you intend to use your system.

NAND Flash flash_erase & nandwrite (from mtd-utils)

UBIFS mkfs.ubifs (from mtd-utils)

TAR archives You may either use GNU tar or Busybox tar.

If you intend to use Busybox tar, make sure format autodetection and also the compression formats you use are enabled:

- `CONFIG_FEATURE_TAR_AUTODETECT=y`
- `CONFIG_FEATURE_TAR_LONG_OPTIONS=y`
- select needed `CONFIG_FEATURE_SEAMLESS_*=y` options

ext4 mkfs.ext4 (from e2fsprogs)

vfat mkfs.vfat (from dosfstools)

Depending on the bootloader you use on your target, RAUC also needs the right tool to interact with it:

Barebox barebox-state (from dt-utils)

U-Boot fw_setenv/fw_getenv (from u-boot)

GRUB grub-editenv

EFI efibootmgr

Note that for running `rauc info` on the target (as well as on the host), you also need to have the `unsquashfs` tool installed.

When using the RAUC casync integration, the `casync` tool must also be available.

6.6 Interfacing with the Bootloader

RAUC provides support for interfacing with different types of bootloaders. To select the bootloader you have or intend to use on your system, set the `bootloader` key in the `[system]` section of your device's `system.conf`.

Note: If in doubt about choosing the right bootloader, we recommend to use barebox as it provides a dedicated boot handling framework, called bootchooser.

To let RAUC handle a bootable slot, you have to mark it as bootable in your `system.conf` and configure the name under which the bootloader identifies this specific slot. This is both done by setting the `bootname` property.

```
[slot.rootfs.0]
...
bootname=system0
```

Amongst others, the bootname property also serves as one way to let RAUC know which slot is currently booted (running). In the following, the different options for letting RAUC detect the currently booted slot are described.

6.6.1 Booted Slot Detection

For RAUC it is quite essential to know from which slot the system is currently running. We will refer this as the *booted slot*. Only reliable detection of the *booted slot* enables RAUC to determine the set of currently inactive slots (that it can safely write to).

If possible, one should always prefer to signal the active slot explicitly from the bootloader to the userspace and RAUC. Only for cases where this explicit way is not possible or unwanted, some alternative approaches of automatically detecting the currently booted slot are implemented in RAUC.

A detailed list of detection mechanism follows.

Identification via Kernel Commandline

RAUC evaluates different kernel commandline parameters in the order they are listed below.

`rauc.slot=` and `rauc.external`

This is the generic way to explicitly set information about which slot was booted by the bootloader. For slots that are handled by a bootloader slot selection mechanism (such as A+B slots) you should specify the slot's configured `bootname`:

```
rauc.slot=system0
```

For special cases where some slots are not handled by the slot selection mechanism (such as a 'last-resort' recovery fallback that never gets explicitly selected) you can also give the name of the slot:

```
rauc.slot=recovery.0
```

When booting from a source not configured in your system.conf (for example from a USB memory stick), you can tell rauc explicitly with the flag `rauc.external`. This means that all slots are known to be inactive and will be valid installation targets. A possible use case for this is to use RAUC during a bootstrapping procedure to perform an initial installation.

`bootchooser.active=`

This is the command-line parameter used by barebox's *bootchooser* mechanism. It will be set automatically by the bootchooser framework and does not need any manual configuration. RAUC compares this against each slot's bootname (not the slot's name as above):

```
bootchooser.active=system0
```

root=

If none of the above parameters is given, the `root=` parameter is evaluated by RAUC to gain information on the currently booted system. The `root=` entry contains the device from which device the kernel (or initramfs) should load the rootfs. RAUC supports parsing different variants for giving these device as listed below.

```
root=/dev/sda1
root=/dev/ubi0_1
```

Giving the plain device name is supported, of course.

Note: The alternative ubi rootfs format with `root=ubi0:volname` is currently unsupported.

```
root=PARTLABEL=abcde
root=PARTUUID=01234
root=UUID=01234
```

Parsing the `PARTLABEL, ``PARTUUID` and `UUID` is supported, which allows referring to a special partition / file system without having to know the enumeration-dependent *sdX* name.

RAUC converts the value to the corresponding `/dev/disk/by-*` symlink name and then to the actual device name.

```
root=/dev/nfs
```

RAUC automatically detects NFS boots (by checking if this parameter is set in the kernel command line). There is no extra slot configuration needed for this as RAUC assumes it is safe to update all available slots in case the currently running system comes from NFS.

6.6.2 Barebox

The Barebox bootloader, which is available for many common embedded platforms, provides a dedicated boot source selection framework, called *bootchooser*, backed by an atomic and redundant storage backend, named *state*.

Barebox state allows you to save the variables required by bootchooser with memory specific storage strategies in all common storage medias, such as block devices, mtd (NAND/NOR), EEPROM, and UEFI variables.

The *Bootchooser* framework maintains information about priority and remaining boot attempts while being configurable on how to deal with them for different strategies.

To enable the Barebox bootchooser support in RAUC, select it in your system.conf:

```
[system]
...
bootloader=barebox
```

Configure Barebox

As mentioned above, Barebox support requires you to have the *bootchooser framework* with *barebox state* backend enabled. In Barebox' Kconfig you can enable this by setting:

```
CONFIG_BOOTCHOOSER=y
CONFIG_STATE=y
CONFIG_STATE_DRV=y
```

To debug and interact with bootchooser and state in Barebox, you should also enable these tools:

```
CONFIG_CMD_STATE=y
CONFIG_CMD_BOOTCHOOSER=y
```

Setup Barebox Bootchooser

The barebox bootchooser framework allows you to specify a number of redundant boot targets that should be automatically selected by an algorithm, based on status information saved for each boot target.

The bootchooser itself can be used as a Barebox boot target. This is where we start by setting the barebox default boot target to *bootchooser*:

```
nv boot.default="bootchooser"
```

Now, when Barebox is initialized it starts the bootchooser logic to select its real boot target.

As a next step, we need to tell bootchooser which boot targets it should handle. These boot targets can have descriptive names which must not equal any of your existing boot targets, we will have a mapping for this later on.

In this example we call the virtual bootchooser boot targets `system0` and `system1`:

```
nv bootchooser.targets="system0 system1"
```

Now connect each of these virtual boot targets to a real Barebox boot target (one of its automagical ones or custom boot scripts):

```
nv bootchooser.system0.boot="nand0.ubi.system0"
nv bootchooser.system1.boot="nand0.ubi.system1"
```

To configure bootchooser to store the variables in Barebox state, you need to configure the `state_prefix`:

```
nv bootchooser.state_prefix="state.bootstate"
```

Beside this very basic configuration variables, you need to set up a set of other general and slot-specific variables.

Warning: It is highly recommended to read the full Barebox bootchooser documentation in order to know about the requirements and possibilities in fine-tuning the behavior according to your needs.

Also make sure to have these `nv` settings in your compiled-in environment, not in your device-local environment.

Setting up Barebox State for Bootchooser

For storing its status information, the bootchooser framework requires a *barebox,state* instance to be set up with a set of variables matching the set of virtual boot targets defined.

To allow loading the state information in a well-defined format both from Barebox and from the kernel, we store the state data format definition in the Barebox devicetree.

Barebox fixups the information into the Linux devicetree when loading the kernel. This assures having a consistent view on the variables in Barebox and Linux.

An example devicetree node for our simple redundant setup will have the following basic structure

```
state {
  bootstate {
    system0 {
    ...
    };
    system1 {
    ...
    };
  };
};
```

In the state node, we set the appropriate compatible to tell the *barebox,state* driver to care for it and define where and how we want to store our data. This will look similar to this:

```
state: state {
        magic = <0x4d433230>;
        compatible = "barebox,state";
        backend-type = "raw";
        backend = <&state_storage>;
        backend-stridesize = <0x40>;
        backend-storage-type = "circular";
        #address-cells = <1>;
        #size-cells = <1>;

        [...]
}
```

where `<&state_storage>` is a phandle to, e.g. an EEPROM or NAND partition.

Important: The devicetree only defines where and in which format the data will be stored. By default, no data will be stored in the deviectree itself!

The rest of the variable set definition will be made in the `bootstate` subnode.

For each virtual boot target handled by state, two uint32 variables `remaining_attempts` and `priority` need to be defined.:

```
bootstate {

        system0 {
                #address-cells = <1>;
                #size-cells = <1>;

                remaining_attempts@0 {
                        reg = <0x0 0x4>;
                        type = "uint32";
                        default = <3>;
                };
                priority@4 {
                        reg = <0x4 0x4>;
                        type = "uint32";
                        default = <20>;
                };
        };
```

(continues on next page)

(continued from previous page)

```
        [...]
};
```

Note: As the example shows, you must also specify some useful default variables the state driver will load in case of uninitialized backend storage.

Additionally one single variable for storing information about the last chosen boot target is required:

```
bootstate {

        [...]

        last_chosen@10 {
                reg = <0x10 0x4>;
                type = "uint32";
        };
};
```

Warning: This example shows only a highly condensed excerpt of setting up Barebox state for bootchooser. For a full documentation on how Barebox state works and how to properly integrate it into your platform see the official Barebox State Framework user documentation as well as the corresponding devicetree binding reference!

You can verify your setup by calling `devinfo state` from Barebox, which would print this for example:

```
barebox@board:/ devinfo state
Parameters:
bootstate.last_chosen: 2 (type: uint32)
bootstate.system0.priority: 10 (type: uint32)
bootstate.system0.remaining_attempts: 3 (type: uint32)
bootstate.system1.priority: 20 (type: uint32)
bootstate.system1.remaining_attempts: 3 (type: uint32)
dirty: 0 (type: bool)
save_on_shutdown: 1 (type: bool)
```

Once you have set up bootchooser properly, you finally need to enable RAUC to interact with it.

Enable Accessing Barebox State for RAUC

For this, you need to specify which (virtual) boot target belongs to which of the RAUC slots you defined. You do this by assigning the virtual boot target name to the slots `bootname` property:

```
[slot.rootfs.0]
...
bootname=system0

[slot.rootfs.1]
...
bootname=system1
```

For writing the bootchooser's state variables from userspace, RAUC uses the tool *barebox-state* from the dt-utils repository.

Note: RAUC requires dt-utils version v2017.03 or later!

Make sure to have this tool integrated on your target platform. You can verify your setup by calling it manually:

```
# barebox-state -d
bootstate.system0.remaining_attempts=3
bootstate.system0.priority=10
bootstate.system1.remaining_attempts=3
bootstate.system1.priority=20
bootstate.last_chosen=2
```

Verify Boot Slot Detection

As detecting the currently booted rootfs slot from userspace and matching it to one of the slots defined in RAUC's `system.conf` is not always trivial and error-prone, Barebox provides an explicit information about which slot it selected for booting adding a *bootchooser.active* key to the commandline of the kernel it boots. This key has the virtual bootchooser boot target assigned. In our case, if the bootchooser logic decided to boot *system0* the kernel commandline will contain:

```
bootchooser.active=system0
```

RAUC uses this information for detecting the active booted slot (based on the slot's *bootname* property).

If the kernel commandline of your booted system contains this line, you have successfully set up bootchooser to boot your slot:

```
$ cat /proc/cmdline
```

6.6.3 U-Boot

To enable handling of redundant booting in U-Boot, manual scripting is required. U-Boot allows storing and modifying variables in its *Environment*. Properly configured, the environment can be accessed both from U-Boot itself as well as from Linux userspace. U-Boot also supports setting up the environment redundantly for atomic modifications.

The default RAUC U-Boot boot selection implementation requires a U-Boot boot script using specific set of variables that are persisted to the environment as stateful slot selection information.

To enable U-Boot support in RAUC, select it in your system.conf:

```
[system]
...
bootloader=uboot
```

Set up U-Boot Boot Script for RAUC

U-Boot as the bootloader needs to decide which slot (partition) to boot. For this decision it needs to read and process some state information set by RAUC or previous boot attempts.

The U-Boot bootloader interface of RAUC will rely on setting the following U-Boot environment variables:

BOOT_ORDER Contains a space-separated list of boot names in the order they should be tried, e.g. `A B`.

BOOT_<bootname>_LEFT Contains the number of remaining boot attempts to perform for the respective slot.

An example U-Boot script for handling redundant A/B boot setups is located in the `contrib/` folder of the RAUC source repository (`contrib/uboot.sh`).

Note: You must adapt the script's boot commands to match the requirements of your platform.

You should integrate your boot selection script as `boot.scr` default boot script into U-Boot.

For this you have to convert it to a U-boot readable default script (`boot.scr`) first:

```
mkimage -A arm -T script -C none -n "Boot script" -d <path-to-input-script> boot.scr
```

If you place this on a partition next to U-Boot, it will use it as its boot script.

For more details, refer the U-Boot Scripting Capabilities chapter in the U-Boot user documentation.

The example script uses the names `A` and `B` as the `bootname` for the two different boot targets. These names need to be set in your system.conf as the `bootname` of the respective slots. The resulting boot attempts variables will be `BOOT_A_LEFT` and `BOOT_B_LEFT`. The `BOOT_ORDER` variable will contain `A B` if `A` is the primary slot or `B A` if `B` is the primary slot to boot.

Note: For minor changes in boot logic or variable names simply change the boot script and/or the RAUC system.conf `bootname` settings. If you want to implement a fully different behavior, you might need to modify the `uboot_set_state()` and `uboot_set_primary()` functions in `src/bootchooser.c` of RAUC.

Setting up the (Fail-Safe) U-Boot Environment

The U-Boot environment is used to store stateful boot selection information and serves as the interface between userspace and bootloader. The information stored in the environment needs to be preserved, even if the bootloader should be updated. Thus the environment should be placed outside the bootloader partition!

The storage location for the environment can be controlled with `CONFIG_ENV_IS_IN_*` U-Boot Kconfig options like `CONFIG_ENV_IS_IN_FAT` or `CONFIG_ENV_IS_IN_MMC`. You may either select a different storage than your bootloader, or a different location/partition/volume on the same storage.

For fail-safe (atomic) updates of the environment, U-Boot can use redundant environments that allow to write to one copy while keeping the other as fallback if writing fails, e.g. due to sudden power cut.

In order to enable redundant environment storage, you have to additionally set in your U-Boot config:

```
CONFIG_SYS_REDUNDAND_ENVIRONMENT=y
CONFIG_ENV_SIZE=<size-of-env>
CONFIG_ENV_OFFSET=<offset-in-device>
CONFIG_ENV_OFFSET_REDUND=<copy-offset-in-device>
```

Note: Above switches refer to U-Boot >= v2020.01.

Refer to U-Boot source code and README for more details on this.

Enable Accessing U-Boot Environment from Userspace

To enable reading and writing of the U-Boot environment from Linux userspace, you need to have:

- U-Boot target tools `fw_printenv` and `fw_setenv` available on your devices rootfs.

- Environment configuration file `/etc/fw_env.config` in your target root filesystem.

See the corresponding HowTo section from the U-Boot documentation for more details on how to set up the environment config file for your device.

Example: Setting up U-Boot Environment on eMMC/SD Card

For this example we assume a simple redundancy boot partition layout with a bootloader partition and two rootfs partitions.

Another additional partition we use exclusively for storing the environment.

Note: It is not strictly required to have the env on an actual MBR/GPT partition, but we use this here as it better protects against accidentially overwriting relevant data of other partitions.

Parition table (excerpt with partition offsets):

```
/dev/mmcblk0p1 StartLBA:   8192 -> u-boot etc.
/dev/mmcblk0p2 StartLBA: 114688 -> u-boot environment
/dev/mmcblk0p3 StartLBA: 139264 -> rootfs A
/dev/mmcblk0p4 StartLBA: 475136 -> rootfs B
```

We enable redundant environment and storage in MMC (not in vfat/ext4 partition) in the u-boot config:

```
CONFIG_SYS_REDUNDAND_ENVIRONMENT=y
CONFIG_ENV_IS_IN_MMC=y
```

The default should be to use mmc device 0 and HW partition 0. Since U-Boot 2020.10.0 we can set this also explicitly if required:

```
CONFIG_SYS_MMC_ENV_DEV=0
CONFIG_SYS_MMC_ENV_PART=0
```

Important: With `CONFIG_SYS_MMC_ENV_PART` we can specify a eMMC HW partition only, not an MBR/GPT partition! HW partitions are e.g. 0=user data area, 1=boot partition.

Then we must specify the env storage size and its offset relative to the currently used device. Here the device is the eMMC user data area (or SD Card). For placing the content in partition 2 now, we must calculate the offset as `offset=hex(n sector * 512 bytes/sector)`. With `n=114688` (start of /dev/mmcblk0p2 according to above partition table) we get an offset of `0x3800000`. As size we pick `0x4000` (16kB) here. The offset of the redundant copy must be the offset of the first copy + size of first copy. This results in:

```
CONFIG_ENV_SIZE=0x4000
CONFIG_ENV_OFFSET=0x3800000
CONFIG_ENV_OFFSET_REDUND=0x3804000
```

Finally, we need to configure userspace to access the same location. This can be referenced directly by its partition device name (/dev/mmcblk0p2) in the `/etc/fw_env.config`:

```
/dev/mmcblk0p2 0x0000 0x4000
/dev/mmcblk0p2 0x4000 0x4000
```

6.6.4 GRUB

```
[system]
...
bootloader=grub
```

To enable handling of redundant booting in GRUB, manual scripting is required.

The GRUB bootloader interface of RAUC uses the GRUB environment variables `<bootname>_OK`, `<bootname>_TRY` and `ORDER`.

An exemplary GRUB configuration for handling redundant boot setups is located in the `contrib/` folder of the RAUC source repository (`grub.conf`). As the GRUB shell only has limited support for scripting, this example uses only one try per enabled slot.

To enable reading and writing of the GRUB environment, you need to have the tool `grub-editenv` available on your target.

By default RAUC expects the grubenv file to be located at `/boot/grub/grubenv`, you can specify a custom directory by passing `grubenv=/path/to/grubenv` in your system.conf `[system]` section.

Make sure that the grubenv file is located outside your redundant rootfs partitions as the rootfs needs to be exchangeable without affecting the environment content. For UEFI systems, a proper location would be to place it on the EFI partition, e.g. at `/EFI/BOOT/grubenv`. The same partition can also be used for your `grub.cfg` (which could be placed at `/EFI/BOOT/grub.cfg`).

6.6.5 EFI

For x86 systems that directly boot via EFI/UEFI, RAUC supports interaction with EFI boot entries by using the *efibootmgr* tool. To enable EFI bootloader support in RAUC, write in your `system.conf`:

```
[system]
...
bootloader=efi
```

To set up a system ready for pure EFI-based redundancy boot without any further bootloader or initramfs involved, you have to create an appropriate partition layout and matching boot EFI entries.

Assuming a simple A/B redundancy, you would need:

- 2 redundant EFI partitions holding an EFI stub kernel (e.g. at `EFI/LINUX/BZIMAGE.EFI`)
- 2 redundant rootfs partitions

To create boot entries for these, use the efibootmgr tool:

```
efibootmgr --create --disk /dev/sdaX --part 1 --label "system0" --loader␣
↪\\EFI\\LINUX\\BZIMAGE.EFI --unicode "root=PARTUUID=<partuuid-of-part-1>"
efibootmgr --create --disk /dev/sdaX --part 2 --label "system1" --loader␣
↪\\EFI\\LINUX\\BZIMAGE.EFI --unicode "root=PARTUUID=<partuuid-of-part-2>"
```

where you replace /dev/sdaX with the name of the disk you use for redundancy boot, `<partuuid-of-part-1>` with the PARTUUID of the first rootfs partition and `<partuuid-of-part-2>` with the PARTUUID of the second rootfs partition.

You can inspect and verify your settings by running:

```
efibootmgr -v
```

In your `system.conf`, you have to list both the EFI partitions (each containing one kernel) as well as the rootfs partitions. Make the first EFI partition a child of the first rootfs partition and the second EFI partition a child of the second rootfs partition to have valid slot groups. Set the rootfs slot bootnames to those we have defined with the `--label` argument in the `efibootmgr` call above:

```
[slot.efi.0]
device=/dev/sdX1
type=vfat
parent=rootfs.0

[slot.efi.1]
device=/dev/sdX2
type=vfat
parent=rootfs.1

[slot.rootfs.0]
device=/dev/sdX3
type=ext4
bootname=system0

[slot.rootfs.1]
device=/dev/sdX4
type=ext4
bootname=system1
```

6.6.6 Custom

If none of the previously mentioned approaches can be applied on the system, RAUC also offers the possibility to use customization scripts or applications as bootloader backend.

To enable the custom bootloader backend support in RAUC, select it in your *system.conf*:

```
[system]
...
bootloader=custom
```

Configure custom bootloader backend

The custom bootloader backed based on a handler that is called to get the desired information or set the appropriate configuration of the custom bootloader environment.

To register the custom bootloader backend handler, assign your handler to the `bootloader-custom-backend` key in section `handlers` in your *system.conf*:

```
[handlers]
...
bootloader-custom-backend=custom-bootloader-script
```

Custom bootloader backend interface

According to *Boot Slot Selection* the custom bootloader handler is called by RAUC to trigger the following actions:

- get the primary slot
- set the primary slot

- get the boot state
- set the boot state

To get the primary slot, the handler is called with the argument `get-primary`. The handler must output the current primary slot's bootname on the *stdout*, and return `0` on exit, if no error occurred. In case of failure, the handler must return with non-zero value. Accordingly, in order to set the primary slot, the custom bootloader handler is called with argument `set-primary <slot.bootname>` where `<slot.bootname>` matches the `bootname=` key defined for the respective slot in your *system.conf*. If the set was successful, the handler must also return with a `0`, otherwise the return value must be non-zero.

In addition to the primary slot, RAUC must also be able to determine the boot state of a specific slot. RAUC determines the necessary boot state by calling the custom bootloader handler with the argument `get-state <slot.bootname>`. Whereupon the handler has to output the state `good` or `bad` to *stdout* and exit with the return value `0`. If the state cannot be determined or another error occurs, the custom bootloader handler must exit with non-zero return value. To set the boot state to the desire slot, the handler is called with argument `set-state <slot.bootname> <state>`. As already mentioned in the paragraph above, the `<slot.bootname>` matches the `bootname=` key defined for the respective slot in your *system.conf*. The `<state>` argument corresponds to one of the following values:

- `good` if the last start of the slot was successful or
- `bad` if the last start of the slot failed.

The return value must be `0` if the boot state was set successfully, or non-zero if an error occurred.

6.7 Init System and Service Startup

There are several ways to run the RAUC service on your target. The recommended way is to use a systemd-based system and allow to start RAUC via D-Bus activation.

You can start the RAUC service manually by executing:

```
$ rauc service
```

6.7.1 Systemd Integration

When building RAUC, a default systemd `rauc.service` file will be generated in the `data/` folder.

Depending on your configuration `make install` will place this file in one of your system's service file folders.

It is a good idea to wait for the system to be fully started before marking it as successfully booted. In order to achieve this, a smart solution is to create a systemd service that calls `rauc status mark-good` and use systemd's dependency handling to assure this service will not be executed before all relevant other services came up successfully. It could look similar to this:

```
[Unit]
Description=RAUC Good-marking Service
ConditionKernelCommandLine=|bootchooser.active
ConditionKernelCommandLine=|rauc.slot

[Service]
ExecStart=/usr/bin/rauc status mark-good

[Install]
WantedBy=multi-user.target
```

6.8 D-Bus Integration

The *D-Bus* interface RAUC provides makes it easy to integrate it into your customapplication. In order to allow sending data, make sure the D-Bus config file `de.pengutronix.rauc.conf` from the `data/` dir gets installed properly.

To only start RAUC when required, using D-Bus activation is a smart solution. In order to enable D-Bus activation, properly install the D-Bus service file `de.pengutronix.rauc.service` from the `data/` dir.

6.9 Watchdog Configuration

Detecting system hangs during runtime requires to have a watchdog and to have the watchdog configured and handled properly. Systemd provides a sophisticated watchdog multiplexing and handling allowing you to configure separate timeouts and handlings for each of your services.

To enable it, you need at least to have these lines in your systemd configuration:

```
RuntimeWatchdogSec=20
ShutdownWatchdogSec=10min
```

6.10 Yocto

Yocto support for using RAUC is provided by the meta-rauc layer.

The layer supports building RAUC both for the target as well as as a host tool. With the bundle.bbclass it provides a mechanism to specify and build bundles directly with the help of Yocto.

For more information on how to use the layer, also see the layer's README file.

6.10.1 Target System Setup

Add the *meta-rauc* layer to your setup:

```
git submodule add git@github.com:rauc/meta-rauc.git
```

Add the RAUC tool to your image recipe (or package group):

```
IMAGE_INSTALL_append = "rauc"
```

Append the RAUC recipe from your BSP layer (referred to as *meta-your-bsp* in the following) by creating a `meta-your-bsp/recipes-core/rauc/rauc_%.bbappend` with the following content:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

Write a `system.conf` for your board and place it in the folder you mentioned in the recipe (*meta-your-bsp/recipes-core/rauc/files*). This file must provide a system compatible string to identify your system type, as well as a definition of all slots in your system. By default, the system configuration will be placed in */etc/rauc/system.conf* on your target rootfs.

Also place the appropriate keyring file for your target into the directory added to `FILESEXTRAPATHS` above. Name it either `ca.cert.pem` or additionally specify the name of your custom file by setting `RAUC_KEYRING_FILE`. If multiple keyring certificates are required on a single system, create a keyring directory containing each certificate.

Note: For information on how to create a testing / development key/cert/keyring, please refer to scripts/README in meta-rauc.

For a reference of allowed configuration options in system.conf, see *System Configuration File*. For a more detailed instruction on how to write a system.conf, see *RAUC System Configuration*.

6.10.2 Using RAUC on the Host System

The RAUC recipe allows to compile and use RAUC on your host system. Having RAUC available as a host tool is useful for debugging, testing or for creating bundles manually. For the preferred way of creating bundles automatically, see the chapter *Bundle Generation*. In order to compile RAUC for your host system, simply run:

```
bitbake rauc-native
```

This will place a copy of the RAUC binary in `tmp/deploy/tools` in your current build folder. To test it, try:

```
tmp/deploy/tools/rauc --version
```

6.10.3 Bundle Generation

Bundles can be created either manually by building and using RAUC as a native tool, or by using the `bundle.bbclass` that handles most of the basic steps, automatically.

First, create a bundle recipe in your BSP layer. A possible location for this could be `meta-your-bsp/recipes-core/bundles/update-bundle.bb`.

To create your bundle you first have to inherit the bundle class:

```
inherit bundle
```

To create the manifest file, you may either use the built-in class mechanism, or provide a custom manifest.

For using the built-in bundle generation, you need to specify some variables:

RAUC_BUNDLE_COMPATIBLE Sets the compatible string for the bundle. This should match the compatible you specified in your `system.conf` or, more generally, the compatible of the target platform you intend to install this bundle on.

RAUC_BUNDLE_SLOTS Use this to list all slot classes for which the bundle should contain images. A value of `"rootfs appfs"` for example will create a manifest with images for two slot classes; rootfs and appfs.

RAUC_BUNDLE_FORMAT Use this to choose the *Bundle Formats* for the generated bundle. It currently defaults to `plain`, but you should use `verity` if possible.

RAUC_SLOT_<slotclass> For each slot class, set this to the image (recipe) name which builds the artifact you intend to place in the slot class.

RAUC_SLOT_<slotclass>[type] For each slot class, set this to the *type* of image you intend to place in this slot. Possible types are: `image` (default), `kernel`, `boot`, or `file`.

Note: For a full list of supported variables, refer to *classes/bundle.bbclass* in meta-rauc.

A minimal bundle recipe, such as *core-bundle-minimal.bb* that is contained in meta-rauc will look as follows:

```
inherit bundle

RAUC_BUNDLE_COMPATIBLE ?= "Demo Board"

RAUC_BUNDLE_SLOTS ?= "rootfs"

RAUC_BUNDLE_FORMAT ?= "verity"

RAUC_SLOT_rootfs ?= "core-image-minimal"
```

To be able to build a signed image of this, you also need to configure `RAUC_KEY_FILE` and `RAUC_CERT_FILE` to point to your key and certificate files you intend to use for signing. You may set them either from your bundle recipe or any global configuration (layer, site.conf, etc.), e.g.:

```
RAUC_KEY_FILE = "${COREBASE}/meta-<layername>/files/development-1.key.pem"
RAUC_CERT_FILE = "${COREBASE}/meta-<layername>/files/development-1.cert.pem"
```

Note: For information on how to create a testing / development key/cert/keyring, please refer to *scripts/README* in meta-rauc.

Based on this information, a call of:

```
bitbake core-bundle-minimal
```

will build all required images and generate a signed RAUC bundle from this. The created bundle can be found in `${DEPLOY_DIR_IMAGE}` (defaults to `tmp/deploy/images/<machine>` in your build directory).

6.11 PTXdist

Note: RAUC support in PTXdist is available since version 2017.04.0.

6.11.1 Integration into Your RootFS Build

To enable building RAUC for your target, set:

```
CONFIG_RAUC=y
```

in your ptxconfig (by selecting `RAUC` via `ptxdist menuconfig`).

You should also customize the compatible RAUC uses for your system. To do this, set `PTXCONF_RAUC_COMPATIBLE` to a string that uniquely identifies your device type. The default value will be `"${PTXCONF_PROJECT_VENDOR}\ ${PTXCONF_PROJECT}"`.

Place your system configuration file in `$(PTXDIST_PLATFORMCONFIGDIR)/projectroot/etc/rauc/system.conf` to let the RAUC package install it into the rootfs you build.

Note: PTXdist versions since 2020.06.0 use their code signing infrastructure for keyring creation. See PTXdist's Managing Certificate Authority Keyrings for different scenarios (refer to RAUC's *CA Configuration*). Previous

PTXdist versions expected the keyring in `$(PTXDIST_PLATFORMCONFIGDIR)/projectroot/etc/rauc/ca.cert.pem`. The keyring is installed into the rootfs to `/etc/rauc/ca.cert.pem`.

If using systemd, the recipes install both the default `systemd.service` file for RAUC as well as a `rauc-mark-good.service` file. This additional good-marking-service runs after user space is brought up and notifies the underlying bootloader implementation about a successful boot of the system. This is typically used in conjunction with a boot attempts counter in the bootloader that is decremented before starting the system and reset by *rauc status mark-good* to indicate a successful system startup.

6.11.2 Create Update Bundles from your RootFS

To enable building RAUC bundles, set:

```
CONFIG_IMAGE_RAUC=y
```

in your platformconfig (by using `ptxdist platformconfig`).

This adds a default image recipe for building a RAUC update bundle out of the system's rootfs. As for most image recipes, the *genimage* <*genimage_*> tool is used to configure and generate the update bundle.

PTXdist's default bundle configuration is placed in *config/images/rauc.config*. You may also copy this to your platform directory to use this as a base for custom bundle configuration.

RAUC enforces signing of update bundles. PTXdist versions since 2020.06.0 use its code signing infrastructure for signing and keyring verification. Previous versions expected the signing key in `$(PTXDIST_PLATFORMCONFIGDIR)/config/rauc/rauc.key.pem`.

Once you are done with your setup, PTXdist will automatically create a RAUC update bundle for you during the run of `ptxdist images`. It will be placed under `$(PTXDIST_PLATFORMDIR)/images/update.raucb`.

6.12 Buildroot

Note: RAUC support in Buildroot is available since version 2017.08.0.

To build RAUC using Buildroot, enable `BR2_PACKAGE_RAUC` in your configuration.

6.13 Bundle Format Migration

Migrating from the *plain* to the *verity* *bundle format* should be simple in most cases and can be done in a single update. The high-level functionality of RAUC (certificate checking, update installation, hooks/handlers, ...) is independent of the low-level bundle format.

The required steps are:

- Configure your build system to build RAUC v1.5 (or newer).
- Enable `CONFIG_MD`, `CONFIG_BLK_DEV_DM` and `CONFIG_DM_VERITY` in your kernel configuration. These may already be enabled if you are using dm-verity for verified boot.
- Add a new bundle output configured for the *verity* format by adding the following to the manifest:

```
[bundle]
format=verity
```

Note: For OE/Yocto with an up-to-date meta-rauc, you can choose the bundle format by adding the `RAUC_BUNDLE_FORMAT = "verity"` option in your bundle recipe. The bundle.bbclass will insert the necessary option into the manifest.

For PTXdist or Buildroot with genimage, you can add the manifest option above to the template in your genimage config file.

With these changes, the build system should produce two bundles (one in either format). A *verity* bundle will only be installable on systems that have already received the migration update. A *plain* bundle will be installable on both migrated and unmigrated systems.

You should then test that *both* bundle formats can be installed on a migrated system, as RAUC will now perform additional checks when installing a `plain` bundle to protect against potential modification during installation. This testing should include all bundle sources (USB, network, ...) that you will need in the field to ensure that these new checks don't trigger in your case (which would prohibit further updates).

Note: When installing bundles from a FAT filesystem (for example on a USB memory stick), check that the mount option `fmask` is set to `0022` or `0133`.

When you no longer need to be able to install previously built bundles in the *plain* format, you should also disable it in the `system.conf`:

```
[system]
...
bundle-formats=-plain
...
```

If you later need to support downgrades, you can use `rauc extract` and `rauc bundle` to convert a *plain* bundle to a *verity* bundle, allowing installation to systems that have already been migrated.

CHAPTER 7

Advanced Topics

7.1 Security

The RAUC bundle format consists of the images and a manifest, contained in a SquashFS image. The SquashFS is followed by a public key signature over the full image. The signature is stored (together with the signer's certificate) in the CMS (Cryptographic Message Syntax, see RFC5652) format. Before installation, the signer certificate is verified against the keyring(s) already stored on the system and the signers pulic key is then used to verify the bundle signature.

We selected the CMS to avoid designing and implementing our own custom security mechanism (which often results in vulnerabilities). CMS is well proven in S/MIME and has widely available implementations, while supporting simple as well as complex PKI use-cases (certificate expiry, intermediate CAs, revocation, algorithm selection, hardware security modules...) without additional complexity in RAUC itself.

RAUC uses OpenSSL as a library for signing and verification of bundles. A PKI with intermediate CAs for the unit tests is generated by the `test/openssl-ca.sh` shell script available from GitHub, which may also be useful as an example for creating your own PKI.

In the following sections, general CA configuration, some use-cases and corresponding PKI setups are described.

7.1.1 CA Configuration

OpenSSL uses an `openssl.cnf` file to define paths to use for signing, default parameters for certificates and additional parameters to be stored during signing. Configuring a CA correctly (and securely) is a complex topic and obviously exceeds the scope of this documentation. As a starting point, the OpenSSL manual pages (especially ca, req, x509, cms, verify and config) and Stefan H. Holek's pki-tutorial are useful.

Certificate Key Usage Attributes

By default (for backwards compatibility reasons), RAUC does not check the certificate's key usage attributes. When not using a stand-alone PKI for RAUC, it can be useful to enable checking via the `check-purpose` configuration option to allow only specific certificates for bundle installation.

When using OpenSSL to create your certificates, the key usage attributes can be configured in the X.509 V3 extension sections in your OpenSSL configuration file. The extension configuration section to be used by `openssl ca` is selected via the `-extensions` argument. For example, RAUC uses a certificate created with the following extensions to test the handling of the *codeSigning extended key usage* attribute:

```
[ v3_leaf_codesign ]
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid:always,issuer:always
basicConstraints = CA:FALSE
extendedKeyUsage=critical,codeSigning
```

As OpenSSL does not (yet) provide a purpose check for code signing, RAUC contains its own implementation, which can be enabled with the *check-purpose=codesign* configuration option. For the leaf (signer) certificate, the *extendedKeyUsage* attribute must exist and contain (at least) the *codeSigning* value. Also, if it has the *keyUsage* attribute, it must contain at least *digitalSignature*. For all other (issuer) certificates in the chain, the *extendedKeyUsage* attribute is optional, but if it is present, it must contain at least the *codeSigning* value.

This means that only signatures using certificates explicitly issued for code signing are accepted for the `codesign` purpose. Also, you can optionally use *extendedKeyUsage* attributes on intermediate CA certificates to limit which ones are allowed to issue code signing certificates.

7.1.2 Single Key

You can use `openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes` to create a key and a self-signed certificate. While you can use RAUC with these, you can't:

- replace expired certificates without updating the keyring
- distinguish between development versions and releases
- revoke a compromised key

7.1.3 Simple CA

By using the (self-signed) root CA only for signing other keys, which are used for bundle signing, you can:

- create one key per developer, with limited validity periods
- revoke keys and ship the CRL (Certificate Revocation List) with an update

With this setup, you can reduce the impact of a compromised developer key.

7.1.4 Separate Development and Release CAs

By creating a complete separate CA and bundle signing keys, you can give only specific persons (or roles) the keys necessary to sign final releases. Each device only has one of the two CAs in its keyring, allowing only installation of the corresponding updates.

While using signing also during development may seem unnecessary, the additional testing of the whole update system (RAUC, bootloader, migration code, ...) allows finding problems much earlier.

7.1.5 Intermediate Certificates

RAUC allows you to include intermediate certificates in the bundle signature that can be used to close the trust chain during bundle signature verification.

To do this, specify the `--intermediate` argument during bundle creation:

```
rauc bundle --intermediate=/path/to/intermediate.ca.pem [...]
```

Note that you can specify the `--intermediate` argument multiple times to include multiple intermediate certificates to your bundle signature.

7.1.6 Resigning Bundles

RAUC allows to replace the signature of a bundle. A typical use case for this is if a bundle that was generated by an autobuilder and signed with a development certificate was tested successfully on your target and should now become a release bundle. For this it needs to be resigned with the release key without modifying the content of the bundle itself.

This is what the `resign` command of RAUC is for:

```
rauc resign --cert=<certfile> --key=<keyfile> --keyring=<keyring> <input-bundle>
↪<output-bundle>
```

It verifies the bundle against the given keyring, strips the old signature and attaches a new one based on the key and cert files provided. If the old signature is no longer valid, you can use the `--no-verify` argument to disable verification.

Switching the Keyring – SPKI hashes

When switching from a development to a release signature, it is typically required to also equip the rootfs with a different keyring file.

While the development system should accept both development and release certificates, the release system should accept only release certificates.

One option to perform this exchange without having to build a new rootfs would be to include both a keyring for the development case as well as a keyring for the release case.

Doing this would be possible in a slot's post-install hook, for example. Depending on whether the bundle to install was signed with a development or a release certificate, either the production or development keyring will be copied to the location where RAUC expects it to be.

To allow comparing hashes, RAUC generates SPKI hashes (i.e. hashes over the entire public key information of a certificate) out of each signature contained in the bundle's trust chain. The SPKI hashes are invariant over changes in signature meta data (such as the validity dates) while allowing to securely compare the certificate ownership.

A simple call of `rauc info` will list the SPKI hashes for each certificate contained in the validated trust chain:

```
Certificate Chain:
 0 Subject: /O=Test Org/CN=Test Org Release-1
   Issuer: /O=Test Org/CN=Test Org Provisioning CA Release
   SPKI sha256:␣
↪94:67:AB:31:08:04:3D:2D:62:D5:EE:58:D6:2F:86:7A:F2:77:94:29:9B:46:11:00:EC:D4:7B:1B:1D:42:8E:5A
 1 Subject: /O=Test Org/CN=Test Org Provisioning CA Release
   Issuer: /O=Test Org/CN=Test Org Provisioning CA Root
   SPKI sha256:␣
↪47:D4:9D:73:9B:11:FB:FD:AB:79:2A:07:36:B7:EF:89:3F:34:5F:D4:9B:F3:55:0F:C1:04:E7:CC:2F:32:DB:11
 2 Subject: /O=Test Org/CN=Test Org Provisioning CA Root
   Issuer: /O=Test Org/CN=Test Org Provisioning CA Root
   SPKI sha256:␣
↪00:34:F8:FE:5A:DC:3B:0D:FE:64:24:07:27:5D:14:4D:E2:39:8C:68:CC:9A:86:DD:67:03:D7:15:11:16:B4:4E
```

A post-install hook instead can access the SPKI hashes via the environment variable `RAUC_BUNDLE_SPKI_HASHES` that will be set by RAUC when invoking the hook script. This variable will contain a space-separated list of the hashes in the same order they are listed in `rauc info`. This list can be used to define a condition in the hook for either installing one or the other keyring file on the target.

Example hook shell script code for above trust chain:

```
case "$1" in

      [...]

      slot-post-install)

              [...]

              # iterate over trust chain SPKI hashes (from leaf to root)
              for i in $RAUC_BUNDLE_SPKI_HASHES; do
                      # Test for development intermediate certificate
                      if [ "$i" ==
↪"46:9E:16:E2:DC:1E:09:F8:5B:7F:71:D5:DF:D0:A4:91:7F:FE:AD:24:7B:47:E4:37:BF:76:21:3A:38:49:89:5B
↪" ]; then
                              echo "Activating development key chain"
                              mv "$RAUC_SLOT_MOUNT_POINT/etc/rauc/devel-keyring.pem" "
↪$RAUC_SLOT_MOUNT_POINT/etc/rauc/keyring.pem"
                              break
                      fi
                      # Test for release intermediate certificate
                      if [ "$i" ==
↪"47:D4:9D:73:9B:11:FB:FD:AB:79:2A:07:36:B7:EF:89:3F:34:5F:D4:9B:F3:55:0F:C1:04:E7:CC:2F:32:DB:11
↪" ]; then
                              echo "Activating release key chain"
                              mv "$RAUC_SLOT_MOUNT_POINT/etc/rauc/release-keyring.pem
↪" "$RAUC_SLOT_MOUNT_POINT/etc/rauc/keyring.pem"
                              break
                      fi
              done
              ;;

      [...]
esac
```

7.1.7 PKCS#11 Support

RAUC can use certificates and keys which are stored in a PKCS#11-supporting smart-card, USB token (such as a YubiKey) or Hardware Security Module (HSM). For all commands which need create a signature `bundle`, `convert` and `resign`, PKCS#11 URLs can be used instead of filenames for the `--cert` and `--key` arguments.

For example, a bundle can be signed with a certificate and key available as `pkcs11:token=rauc;object=autobuilder-1`:

```
rauc bundle \
  --cert='pkcs11:token=rauc;object=autobuilder-1' \
  --key='pkcs11:token=rauc;object=autobuilder-1' \
  <input-dir> <output-file>
```

Note: Most PKCS#11 implementations require a PIN for signing operations. You can either enter the PIN interac-

tively as requested by RAUC or use the `RAUC_PKCS11_PIN` environment variable to specify the PIN to use.

When working with PKCS#11, some tools are useful to configure and show your tokens:

p11-kit p11-kit is an abstraction layer which provides access to multiple PKCS#11 modules.

It contains `p11tool`, which is useful to see available tokens and objects (keys and certificates) and their URLs:

```
$ p11tool --list-tokens
...
Token 5:
        URL: pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;
↪serial=9f03d1aaed92ef58;token=rauc
        Label: rauc
        Type: Generic token
        Manufacturer: SoftHSM project
        Model: SoftHSM v2
        Serial: 9f03d1aaed92ef58
        Module: /usr/lib/softhsm/libsofthsm2.so
$ p11tool --login --list-all pkcs11:token=rauc
Token 'rauc' with URL 'pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;
↪serial=9f03d1aaed92ef58;token=rauc' requires user PIN
Enter PIN: ****
Object 0:
        URL: pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;
↪serial=9f03d1aaed92ef58;token=rauc;id=%01;object=autobuilder-1;type=public
        Type: Public key
        Label: autobuilder-1
        Flags: CKA_WRAP/UNWRAP;
        ID: 01

Object 1:
        URL: pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;
↪serial=9f03d1aaed92ef58;token=rauc;id=%01;object=autobuilder-1;type=private
        Type: Private key
        Label: autobuilder-1
        Flags: CKA_WRAP/UNWRAP; CKA_PRIVATE; CKA_SENSITIVE;
        ID: 01

Object 2:
        URL: pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;
↪serial=9f03d1aaed92ef58;token=rauc;id=%01;object=autobuilder-1;type=cert
        Type: X.509 Certificate
        Label: autobuilder-1
        ID: 01
```

OpenSC OpenSC is the standard open source framework for smart card access.

It provides `pkcs11-tool`, which is useful to prepare a token for usage with RAUC. It can list, read/write objects, generate keypairs and more.

libp11 libp11 is an engine plugin for OpenSSL, which allows using keys on PKCS#11 tokens with OpenSSL.

It will automatically use p11-kit (if available) to access all configured PKCS#11 modules.

Note: If you cannot use p11-kit, you can also use the `RAUC_PKCS11_MODULE` environment variable to select the PKCS#11 module.

SoftHSM2 SoftHSM2 is software implementation of a HSM with a PKCS#11 interface.

It is used in the RAUC test suite to emulate a real HSM and can also be used to try the PKCS#11 functionality in RAUC without any hardware. The `prepare_softhsm2` shell function in `test/rauc.t` can be used as an example on how to initialize SoftHSM2 token.

7.1.8 Protection Against Concurrent Bundle Modification

As the `plain` *bundle format* consists of a squashfs image with an appended CMS signature, RAUC must check the signature before accessing the squashfs. If an unprivileged process can manipulate the squashfs part of the bundle after the signature has been checked, it could use this to elevate its privileges.

The `verity` format is not affected by this problem, as the kernel checks the squashfs data as it is read.

To mitigate this problem when using the `plain` format, RAUC will check the bundle file for possible issues before accessing the squashfs:

- ownership or permissions that would allow other users to open it for writing
- storage on unsafe filesystems such as FUSE or NFS, where the data is supplied by an untrusted source (the rootfs is explicitly trusted, though)
- storage on a filesystem mounted from a block device with a non-root owner
- existing open file descriptors (via `F_SETLEASE`)

If the check fails, RAUC will attempt to take ownership of the bundle file and removes write permissions. This protects against processes trying to open writable file descriptors from this point on. Then, the checks above a repeated before setting up the loopback device and mounting the squashfs. If this second check fails, RAUC will abort the installation.

If RAUC had to take ownership of the bundle, this change is not reverted after the installation is completed. Note that, if the original user has write access to the containing directory, they can still delete the file.

7.2 Data Storage and Migration

Most systems require a location for storing configuration data such as passwords, ssh keys or application data. When performing an update, you have to ensure that the updated system takes over or can access the data of the old system.

7.2.1 Storing Data in The Root File System

In case of a writable root file system, it often contains additional data, for example cryptographic material specific to the machine, or configuration files modified by the user. When performing the update, you have to ensure that the files you need to preserve are copied to the target slot after having written the system data to it.

RAUC provides support for executing *hooks* from different slot installation stages. For migrating data from your old rootfs to your updated rootfs, simply specify a slot post-install hook. Read the *Hooks* chapter on how to create one.

7.2.2 Using Data Partitions

Often, there are a couple of reasons why you don't want to or cannot store your data inside the root file system:

- You want to keep your rootfs read-only to reduce probability of corrupting it.
- You have a non-writable rootfs such as SquashFS.
- You want to keep your data separated from the rootfs to ease setup, reset or recovery.

In this case you need a separate storage location for your data on a different partition, volume or device.

If the update concept uses full redundant root file systems, there are also good reasons for using a redundant data storage, too. Read below about the possible impact on data migration.

To let your system access the separate storage location, it has to be mounted into your rootfs. Note that if you intend to store configurable system information on your data partition, you have to map the default Linux paths (such as `/etc/passwd`) to your data storage. You can do this by using:

- symbolic links
- bind mounts
- an overlay file system

It depends on the amount and type of data you want to handle which option you should choose.

7.2.3 Application Data Migration

Both a single and a redundant data storage have their advantages and disadvantages. Note when storing data inside your rootfs you will have a redundant setup by design and cannot choose.

The decision about how to set up a configuration storage and how to handle it depends on several aspects:

- May configuration formats change over different application versions?
- Can a new application read (and convert) old data?
- Does your infrastructure allow working on possibly obsolete data?
- Enough storage to store data redundantly?
- ...

The basic advantages and disadvantages a single or a redundant setup implicate are listed below:

	Single Data	Redundant Data
Setup	easy	assure using correct one
Migration	no backup by default	copy on update, migrate
Fallback	tricky (reconvert data?)	easy (old data!)

Managing a `/dev/data` Symbolic Link

For redundant data partitions the active rootfs slot has to mount the correct data partition dynamically. For example with ubifs, a udev ruleset can be used for this:

```
KERNEL=="ubi[0-9]_[0-9]", PROGRAM="/usr/bin/is-parent-active %k", RESULT=="1",␣
↪SYMLINK+="data"
```

This example first determines if ubiX_Y is a data slot with an active parent rootfs slot by calling the script below. Then, the current ubiX_Y partition is bound to /dev/data if the secript retured `1` as its output.

`/usr/bin/is-parent-active` is a simple bash script:

```
#!/bin/bash

ROOTFS_DEV=<determine rootfs by using proc cmdline or mount>
```

(continues on next page)

(continued from previous page)

```
TEST_DEV=<obtain parent rootfs device for currently processed device (%k)>

if [[ $ROOTFS_DEV == $TEST_DEV ]]; then
      echo 1
else
      echo 0
fi
```

With this you can always mount `/dev/data` and get the correct data slot.

7.3 RAUC casync Support

> **Warning:** casync support is still experimental and lacks some unit tests.
>
> When evaluating, make sure to compile a recent casync version from the git for testing.

Using the Content-Addressable Data Synchronization tool *casync* for updating embedded / IoT devices provides a couple of benefits. By splitting and chunking the update artifacts into reusable pieces, casync allows to

- stream remote bundles to the target without occupying storage / NAND
- minimize transferred data for an update by downloading only the delta to the running system
- reduce data storage on server side by eliminating redundancy
- good handling for CDNs due to similar chunk sizes

For a full description of the way casync works and what you can do with it, refer to the blog post by its author Lennart Poettering or visit the GitHub site.

RAUC supports using casync index files instead of complete images in its bundles. This way the real size of the bundle comes down to the size of the index files required for referring to the individual chunks. The real image data contained in the individual chunks can be stored in one single repository, for a whole systems with multiple images as well as for multiple systems in different versions, etc. This makes the approach quite flexible.

7.3.1 Creating casync Bundles

Creating RAUC bundles with casync index files is a bit different from creating 'conventional' bundles. While the bundle format remains the same and you could also mix conventional and casync-based bundles, creating these bundles is not straight forward when using common embedded build systems such as Yocto, PTXdist or buildroot.

Because of this, we decided use a two-step process for creating casync RAUC bundles:

1. Create 'conventional' RAUC bundle
2. Convert to casync-based RAUC bundle

RAUC provides a command for creating casync-based bundles from 'conventional' bundles. Simply call:

```
rauc convert --cert=<certfile> --key=<keyfile> --keyring=<keyring> conventional-
↪bundle.raucb casync-bundle.raucb
```

The conversion process will create two new artifacts:

1. The converted bundle *casync-bundle.raucb* with casnyc index files instead of image files
2. A casync chunk store *casync-bundle.castr/* for all bundle images. This is a directory with chunks grouped by subfolders of the first 4 digits of their chunk ID.

7.3.2 Installing casync Bundles

The main difference between installing conventional bundles and bundles that contain casync index files is that RAUC requires access to the remote casync chunk store during installation of the bundle.

Due to the built-in network support of both casync and RAUC, it is possible to directly give a network URL as the source of the bundle:

```
rauc install https://server.example.com/deploy/bundle-20180112.raucb
```

By default, RAUC will assume the corresponding casync chunk store is located at the same location as the bundle (with the `.castr` extension instead of `.raucb`), in this example at `https://server.example.com/deploy/bundle-20180112.castr`. The default location can also be configured in the system config to point to a generic location that is valid for all installations.

When installing a bundle, the casync implementation will automatically handle the chunk download via an unprivileged helper binary.

Reducing Download Size – Seeding

Reducing the amount of data to be transferred over slow connections is one of the main goals of using casync for updating. Casync splits up the images or directory trees it handles into reusable chunks of similar size. Doing this both on the source as well as on the destination side allows comparing the hashes of the resulting chunks to know which parts are different.

When we update a system, we usually do not change its entire file tree, but only update a few libraries, the kernel, the application, etc. Thus, most of the data can be retrieved from the currently active system and does not need to be fetched via the network.

For each casync image that RAUC extracts to the target slot, it determines an appropriate seed. This is normally a redundant slot of the same class as the target slot but from the currently booted slot group.

Note: Depending on your targets processing and storage speed, updating slots with casync can be a bit slower than conventional updates, because casync first has to process the entire seed slot to calculate the seed chunks. After this is done it will start writing the data and fetch missing chunks via the network.

7.4 Handling Board Variants With a Single Bundle

If you have hardware variants that require installing different images (e.g. for the kernel or for an FPGA bitstream), but have other slots that are common (such as the rootfs) between all hardware variants, RAUC allows you to put multiple different variants of these images in the same bundle. RAUC calls this feature 'image variants'.

If you want to make use of image variants, you first of all need to say which variant your specific board is. You can do this in your `system.conf` by setting exactly one of the keys `variant-dtb`, `variant-file` or `variant-name`.

```
[system]
...
variant-dtb=true
```

The `variant-dtb` is a boolean that allows (on device-tree based boards) to use the systems compatible string as the board variant.

```
[system]
...
variant-file=/path/to/file
```

A more generic alternative is the `variant-file` key. It allows to specify a file that will be read to obtain the variant name. Note that the content of the file should be a simple string without any line breaks. A typical use case would be to generate this file (in `/run`) during system startup from a value you obtained from your bootloader. Another use case is to have a RAUC post-install hook that copies this file from the old system to the newly updated one.

```
[system]
...
variant-name=myvariant-name
```

A third variant to specify the systems variant is to give it directly in your system.conf. This method is primary meant for testing, as this prevents having a generic rootfs image for all variants!

In your manifest, you can specify variants of an image (e.g. the kernel here) as follows:

```
[image.kernel.variant-1]
filename=variant1.img
...

[image.kernel.variant-2]
filename=variant1.img
...
```

It is allowed to have both a specific variant as well as a default image in the same bundle. If a specific variant of the image is available, it will be used on that system. On all other systems, the default image will be used instead.

If you have a specific image variant for one of your systems, it is mandatory to also have a default or specific variant for the same slot class for any other system you intend to update. RAUC will report an error if for example a bootloader image is only present for variant A when you try to install on variant B. This should prevent bricking your device by unintentional partial updates.

7.5 Manually Writing Images to Slots

In order to write an image to a slot without using update mechanics like hooks, slot status etc. use:

```
rauc write-slot <slotname> <image>
```

This uses the correct handler to write the image to the slot. It is useful for development scenarios as well as initial provisioning of embedded boards.

7.6 Updating the Bootloader

Updating the bootloader is a special case, as it is a single point of failure on most systems: The selection of which redundant system images should be booted cannot itself be implemented in a redundant component (otherwise there would need to be an even earlier selection component).

Some SoCs contain a fixed firmware or ROM code which already supports redundant bootloaders, possibly integrated with a HW watchdog or boot counter. On these platforms, it is possible to have the selection point before the bootloader, allowing it to be stored redundantly and updated as any other component.

If redundant bootloaders with fallback is not possible (or too inflexible) on your platform, you may instead be able to ensure that the bootloader update is atomic. This doesn't support recovering from a buggy bootloader, but will prevent a non-bootable system caused by an error or power-loss during the update.

Whether atomic bootloader updates can be implemented depends on your SoC/firmware and storage medium.

7.6.1 Update eMMC Boot Partitions

RAUC supports updating eMMC boot partitions (see the JEDEC standard JESD84-B51 for details), one of which gets enabled atomically via configuration registers in the eMMC (*ext_csd registers*).

The required slot type is `boot-emmc`. The device to be specified is expected to be the root device. The boot partitions are derived automatically. A `system.conf` could look like this:

```
[slot.bootloader.0]
device=/dev/mmcblk1
type=boot-emmc
```

Important: A kernel bug may prevent consistent toggling of the eMMC ext_csd boot partition register. Be sure your kernel is >= 4.16-rc7 (resp. >= 4.15.14, >= 4.14.31) or contains this patch: https://www.spinics.net/lists/linux-mmc/msg48271.html

7.6.2 Update Boot Partition in MBR

Some SOCs (like Xilinx ZynqMP) contain a fixed ROM code, which boots from the first partition in the MBR partition table of a storage medium. In order to atomically update the bootloader of such systems, RAUC supports changing the MBR partition table and thus switching between two partitions of the same size - one active boot partition (i.e. the partition is defined in the MBR partition table) and one inactive partition (i.e. there is no entry for it in the MBR partition table) which is used to update the bootloader.

A memory region, where the two partitions are stored has to be defined in the configuration (see below) and initially a boot partition has to exist at either the start of the region or start + size / 2.

Consider the following example layout of a storage medium with a boot partition size of 33 Mbytes:

Start	Size	
0x00000000	512 bytes	MBR
0x00000200	160 Kbytes	Space for state, barebox-environment, ...
0x00028200 0x00028200 0x02128200	66 Mbytes 33 Mbytes 33 Mbytes	MBR switch region containing: - active boot partition (entry in MBR) - inactive boot partition (no entry in MBR)
0x04228200	Remaining size	other partitions (partition table entries 2, 3, 4)

RAUC uses the start address and size defined in the first entry in the MBR partition table, to distinguish between active and inactive boot partition and updates the hidden, inactive partition. After the update the bootloader is switched by changing the first partition entry and writing the whole 512 bytes MBR atomically.

The required slot type is `boot-mbr-switch`. The device to be specified is expected to be the underlying block device. The boot partitions are derived by the definition of the values `region-start` and `region-size`. Both values have to be set in integer decimal bytes and can be post-fixed with K/M/G/T.

A `system.conf` section could look like this:

```
[slot.bootloader.0]
device=/dev/mmcblk1
type=boot-mbr-switch
region-start=164352
region-size=66M
```

7.6.3 Update Boot Partition in GPT

Systems booting via UEFI have a special partition, called the *EFI system partition (ESP)*, which contains the bootloader to be started by the UEFI firmware. Also, some newer ARM SoCs support loading the bootloader directly from a GPT partition.

To allow atomic updates of these partitions, RAUC supports changing the GPT to switch the first GPT partition between the lower and upper halves of a region configured for that purpose. This works similarly to the handling of a MBR boot partition as described in the previous section. It requires RAUC to be compiled with GPT support (`./configure --enable-gpt`) and adds a dependency on libfdisk.

The required slot type is `boot-gpt-switch`. The device to be specified is expected to be the underlying block device. The boot partitions are derived by the definition of the values `region-start` and `region-size`. Both values have to be set in integer decimal bytes and can be post-fixed with K/M/G/T.

To ensure that the resulting GPT entries are well aligned, the region start must be a multiple of the *grain* value (as used by `sfdisk`), which is 1MB by default. Accordingly, the region size must be aligned to twice the *grain* value (to ensure that the start of the upper half is aligned).

Note that RAUC expects that the partition table always points exactly to one of the halves.

A `system.conf` section could look like this:

```
[slot.esp.0]
device=/dev/sda
type=boot-gpt-switch
region-start=1M
region-size=32M
```

7.6.4 Bootloader Update Ideas

The NXP i.MX6 supports up to four bootloader copies when booting from NAND flash. The ROM code will try each copy in turn until it finds one which is readable without uncorrectable ECC errors and has a correct header. By using the trait of NAND flash that interrupted writes cause ECC errors and writing the first page (containing the header) last, the bootloader images can be replaced one after the other, while ensuring that the system will boot even in case of a crash or power failure.

The slot type could be called "boot-imx6-nand" analogous to eMMC.

7.6.5 Considerations When Updating the Bootloader

Booting an old system with a new bootloader is usually not tested during development, increasing the risk of problems appearing only in the field. If you want to address this issue do not add the bootloader to your bundle, but rather use an approach like this:

- Store a copy of the bootloader in the rootfs.
- Use RAUC only to update the rootfs. The combinations to test can be reduced by limiting which old versions are supported by an update.
- Reboot into the new system.
- On boot, before starting the application, check that the current slot is 'sane'. Then check if the installed bootloader is older than the version shipped in the (new) rootfs. In that case:
  - Disable the old rootfs slot and update the bootloader.
  - Reboot
- Start the application.

This way you still have fallback support for the rootfs upgrade and need to test only:

- The sanity check functionality and the bootloader installation when started from old bootloader and new rootfs
- Normal operation when started from new bootloader and new rootfs

The case of new bootloader with old rootfs can never happen, because you disable the old one from the new before installing a new bootloader.

If you need to ensure that you can fall back to the secondary slot even after performing the bootloader update, you should check that the "other" slot contains the same bootloader version as the currently running one during the sanity check. This means that you need to update both slots in turn before the bootloader is updated.

7.7 Updating Sub-Devices

Besides the internal storage, some systems have external components or sub-devices which can be updated. For example:

- Firmware for micro-controllers on modular boards

- Firmware for a system management controller
- FPGA bitstreams (stored in a separate flash)
- Other Linux-based systems in the same enclosure
- Software for third-party hardware components

In many cases, these components have some custom interface to query the currently installed version and to upload an update. They may or may not have internal redundancy or recovery mechanisms as well.

Although it is possible to configure RAUC slots for these and let it call a script to perform the installation, there are some disadvantages to this approach:

- After a fallback to an older version in an A/B scenario, the sub-devices may be running an incompatible (newer) version.
- A modular sub-device may be replaced and still has an old firmware version installed.
- The number of sub-devices may not be fixed, so each device would need a different slot configuration.

Instead, a more robust approach is to store the sub-device firmware in the rootfs and (if needed) update them to the current versions during boot. This ensures that the sub-devices are always running the correct set of versions corresponding to the version of the main application.

If the bootloader falls back to the previous version on the main system, the same mechanism will downgrade the sub-devices as needed. During a downgrade, sub-devices which are running Linux with RAUC in an A/B scenario will detect that the image to be installed already matches the one in the other slot and avoid unnecessary installations.

7.8 Migrating to an Updated Bundle Version

As RAUC undergoes constant development, it might be extended and new features or enhancements will make their way into RAUC. Thus, also the sections and options contained in the bundle manifest may be extended over time.

To assure a well-defined and controlled update procedure, RAUC is rather strict in parsing the manifest and will reject bundles containing unknown configuration options.

But, this does not prevent you from being able to use those new RAUC features on your current system. All you have to do is to perform an *intermediate update*:

- Create a bundle containing a rootfs with the recent RAUC version, but *not* containing the new RAUC features in its manifest.
- Update your system and reboot
- Now you have a system with a recent RAUC version which is able to interpret and appropriately handle a bundle with the latest options

7.9 Software Deployment

When designing your update infrastructure, you must think about how to deploy the updates to your device(s). In general, you have two major options: Deployment via storage media such as USB sticks or network-based deployment.

As RAUC uses signed bundles instead of e.g. trusted connections to enable update author verification, RAUC fully supports both methods with the same technique and you may also use both of them in parallel.

Some influential factors on the method to used can be:

- Do you have network access on the device?

- How many devices have to be updated?
- Who will perform the update?

7.9.1 Deployment via Storage Media

This method is mainly used for decentralized updates of devices without network access (either due to missing infrastructure or because of security concerns).

To handle deployment via storage media, you need a component that detects the plugged-in storage media and calls RAUC to trigger the actual installation.

When using systemd, you could use automount units for detecting plugged-in media and trigger an installation.

7.9.2 Deployment via Deployment Server

Deployment over a network is especially useful when having a larger set of devices to update or direct access to these devices is tricky.

As RAUC focuses on update handling on the target side, it does not provide a deployment server out of the box. But if you do not already have a deployment infrastructure, there a few Open Source deployment server implementations available in the wilderness.

One of it worth being mentioned is hawkBit from the Eclipse IoT project, which also provides some strategies for rollout management for larger-scale device farms.

RAUC hawkBit updater (C)

The rauc-hawkbit-updater is a separate application project developed under the rauc organization umbrella. It aims to provide a ready-to-use bridge between the hawkBit REST DDR API on one side and the RAUC D-Bus API on the other.

For more information visit it on GitHub:

https://github.com/rauc/rauc-hawkbit-updater

The RAUC hawkBit client (python)

As a separate project, the RAUC development team provides a Python-based example application that acts as a hawkBit client via its REST DDI-API while controlling RAUC via D-Bus.

For more information visit it on GitHub:

https://github.com/rauc/rauc-hawkbit

It is also available via pypi:

https://pypi.python.org/pypi/rauc-hawkbit/

CHAPTER 8

Design Checklist

This checklist is intended to help you verify that your design and implementation cover the important corner-cases and details. Even if not all items are ticked off for your system, it's useful to have at least thought about them. Most of these are general considerations and not strictly RAUC specific.

8.1 General

- System compatible is specific enough
- Bundle version policy defined
- Bundle contains all software components
- Bundles are created automatically by a build system
- Bundles use the `verity` *format*
- Bundle *format* `plain` is disabled in `system.conf`
- Bundle deployment mechanism defined (pull or push via the network, from USB/SD, . . .)
- Proper slot status file location(s) defined (preferably central status)

8.2 Slot Layout

- Slot layout provides the desired redundancy
- Complexity vs. simplicity trade-offs understood
- Single points of failure identified and well tested
- Factory disk image includes all slots with default contents
- Appropriate image formats selected (tar or filesystem-image)
- Bootloader uses the same names configured in `system.conf` as `bootname`

- Bootloader update mechanism defined (or declared as fixed)

8.3 Recovery Mechanism

- The initial (factory) boot configuration is correct
- Boot failures are detected by the bootloader
- Booting the same slot is retried the correct number of times (once or more)
- The behavior if one slot fails to boot is defined (fallback to old version or not)
- The behavior if all slots fail to boot is defined (retry or poweroff)

8.3.1 If Using a HW Watchdog for Error Detection

- Watchdog is never disabled before application is ready
- Bootloader distinguishes watchdog resets from normal boot
- Bootloader ensures the watchdog is enabled before starting the kernel
- The watchdog reset reinitializes the whole system (power supplies, storage, SoC, . . .)
- All essential services are monitored by the watchdog

8.3.2 If Not Using a HW Watchdog for Error Detection

- Bootloader detects failed boots via a counter
- Boot counter is reset on a successful boot
- All essential services work before confirming the current boot as successful

8.4 Security

- PKI configured
- Certificate validity periods defined
  - Systems always have correct time *or*
  - Validity period is large enough
- Key revocation tested
  - Updated CRLs can be deployed in time *or*
  - CRLs do not expire
- Key rollover tested
- Separate development and release keys deployed
- Per-user or per-role keys deployed

8.5 Data Migration

- Passwords/SSH keys are preserved during updates
- Shared data is handled correctly during up- and downgrades

CHAPTER 9

Frequently Asked Questions

9.1 Why doesn't the installed system use the whole partition?

The filesystem image installed via RAUC was probably created for a size smaller than the partition on the target device.

Especially in cases where the same bundle will be installed on devices which use different partition sizes, tar archives are preferable to filesystem images. When RAUC installs from a tar archive, it will first create a new filesystem on the target partition, allowing use of the full size.

9.2 Is it possible to use RAUC without D-Bus (Client/Server mode)?

Yes. If you compile RAUC using the `--disable-service` configure option, you will be able to compile RAUC without service mode and without D-Bus support:

```
./configure --disable-service
```

Then every call of the command line tool will be executed directly rather than being forwarded to the RAUC service process running on your machine.

9.3 Why does RAUC not have an ext2 / ext3 file type?

ext4 is the successor of ext3. There is no advantage in using ext3 over ext4.

Some people still tend to select ext2 when they want a file system without journaling. This is not necessary, as one can turn off journaling in ext4, either during creation:

```
mkfs.ext4 -O ^has_journal
```

or later with:

```
tune2fs -O ^has_journal
```

Note that even if there is only an ext4 slot type available, potentially each file system mountable as ext4 should work (with the filename suffix adapted).

9.4 Is the RAUC bundle format forwards/backwards compatible?

The basic bundle format has not changed so far (squashfs containing images and the manifest, with a CMS signature), which means that newer versions can install old bundles. Going forward, any issue with installing old bundles would be considered a bug.

Newer RAUC versions have added features and slot types, though (such as casync, eMMC boot partitions, MBR/GPT partition switching). If you use those features, older versions of RAUC that cannot handle them will refuse to install the bundle. As long as you don't use new features, our intention is that bundles created by newer versions will be installable by older versions.

There are ideas of introducing a new bundle format to allow streaming installation (over the network), but we won't remove support for the original format.

If there are ever reasons that require an incompatible change, you can use a two step migration: You can use an intermediate update to ship a new RAUC binary in a bundle created by the old (compatible) version. Then use the newly installed RAUC binary for the real update.

9.5 Can I use RAUC with a dm-verity-protected partition?

Yes you can, as the offline-generated dm-verity hash tree is simply part of the image that RAUC writes to the partition. To ensure RAUC does not corrupt the partition by executing hooks or writing slot status information, use `type=raw` in the respective slot config and use a global (see *slot status file*) on a separate non-redundant partition with setting `statusfile=</path/to/global.status>`.

9.6 What causes a payload size that is not a multiple of 4kiB?

RAUC versions up to 1.4 had an issue in the casync bundle signature generation, which caused two signatures to be appended. While the squashfs payload size is a multiple of 4kiB, the end of the first signature was not aligned. As RAUC uses the second ("outer") signature during verification, this didn't cause problems. RAUC 1.5 fixed the casync bundle generation and added stricter checks, which rejected the older bundles. In RAUC 1.5.1, this was reduced to a notification message.

To avoid the message, you can recreate the bundle with RAUC 1.5 and newer.

CHAPTER 10

Reference

10.1 System Configuration File

A configuration file located in `/etc/rauc/system.conf` describes the number and type of available slots. It is used to validate storage locations for update images. Each board type requires its special configuration.

This file is part of the root file system.

Note: When changing the configuration file on your running target you need to restart the RAUC service in order to let the changes take effect.

Example configuration:

```
[system]
compatible=FooCorp Super BarBazzer
bootloader=barebox
statusfile=/data/central-status.raucs
bundle-formats=-plain

[keyring]
path=/etc/rauc/keyring.pem

[handlers]
system-info=/usr/lib/rauc/info-provider.sh
post-install=/usr/lib/rauc/postinst.sh

[slot.rootfs.0]
device=/dev/sda0
type=ext4
bootname=system0

[slot.rootfs.1]
device=/dev/sda1
type=ext4
bootname=system1
```

[system] section

compatible A user-defined compatible string that describes the target hardware as specific enough as required to prevent faulty updating systems with the wrong firmware. It will be matched against the `compatible` string defined in the update manifest.

bootloader The bootloader implementation RAUC should use for its slot switching mechanism. Currently supported values (and bootloaders) are `barebox`, `grub`, `uboot`, `efi`, `custom`, `noop`.

bundle-formats This option controls which *bundle formats* are allowed when verifying a bundle. You can either specify them explicitly by using a space-separated list for format names (such as `plain verity`). In this case, any any future changes of the built-in defaults will have no effect.

Alternatively, you can use format names prefixed by `-` or `+` (such as `-plain`) to enable or disable formats relative to the default configuration. This way, formats added in newer releases will be active automatically.

mountprefix Prefix of the path where bundles and slots will be mounted. Can be overwritten by the command line option `--mount`. Defaults to `/mnt/rauc/`.

grubenv Only valid when `bootloader` is set to `grub`. Specifies the path under which the GRUB environment can be accessed.

barebox-statename Only valid when `bootloader` is set to `barebox`. Overwrites the default state `state` to a user-defined state name. If this key not exists, the bootchooser framework searches per default for `/state` or `/aliases/state`.

efi-use-bootnext Only valid when `bootloader` is set to `efi`. If set to `false`, this disables using efi variable `BootNext` for marking a slot primary. This is useful for setups where the BIOS already handles the slot switching on watchdog resets. Behavior defaults to `true` if option is not set.

activate-installed This boolean value controls if a freshly installed slot is automatically marked active with respect to the used bootloader. Its default value is `true` which means that this slot is going to be started the next time the system boots. If the value of this parameter is `false` the slot has to be activated manually in order to be booted, see section *Manually Switch to a Different Slot*.

statusfile This key should be set to point to a central file where slot status information should be stored (e.g. slot-specific metadata, see *Slot Status*). This file must be located on a non-redundant filesystem which is not

overwritten during updates. In most cases, a central status file is preferable to per-slot status files as it allows to store data also for read-only or (temporary) filesystem-less slots. However, if a per-slot status is required as one of the above-noted requirements cannot be met, one can use the value `per-slot` to document this decision. For background compatibility this option is not mandatory and will default to per-slot status files if not set.

`max-bundle-download-size` Defines the maximum downloadable bundle size in bytes, and thus must be a simple integer value (without unit) greater than zero. It overwrites the compiled-in default value of 8 MiB.

`variant-name` String to be used as variant name for this board. If set, neither `variant-file` nor `variant-dtb` must be set. Refer chapter *Handling Board Variants With a Single Bundle* for more information.

`variant-file` File containing variant name for this board. If set, neither `variant-name` nor `variant-dtb` must be set. Refer chapter *Handling Board Variants With a Single Bundle* for more information.

`variant-dtb` If set to `true`, use current device tree compatible as this boards variant name. If set, neither `variant-name` nor `variant-file` must be set. Refer chapter *Handling Board Variants With a Single Bundle* for more information.

[keyring] section

The `keyring` section refers to the trusted keyring used for signature verification. Both `path` and `directory` options can be used together if desired, though only one or the other is necessary to verify the bundle signature.

`path` Path to the keyring file in PEM format. Either absolute or relative to the system.conf file.

`directory` Path to the keyring directory containing one or more certificates. Each file in this directory must contain exactly one certificate in CRL or PEM format. The filename of each certificate must have the form hash.N for a certificate or hash.rN for CRLs; where hash is obtained by `X509_NAME_hash(3)` or the `--hash` option of `openssl(1) x509` or `crl` commands. See documentation in `X509_LOOKUP_hash_dir(3)` for details.

`use-bundle-signing-time=<true/false>` If this boolean value is set to `true` then the bundle signing time is used instead of the current system time for certificate validation.

`check-crl=<true/false>` If this boolean value is set to `true`, RAUC will enable checking of CRLs (Certificate Revocation Lists) stored in the keyring together with the CA certificates. Note that CRLs have an expiration time in their signature, so you need to make sure you don't end up with an expired CRL on your device (which would block further updates).

`check-purpose` This option can be used to set the OpenSSL certificate purpose used during chain verification. Certificates in the chain with incompatible purposes are rejected. Possible values are provided by OpenSSL (`any`, `sslclient`, `sslserver`, `nssslserver`, `smimesign`, `smimeencrypt`) and RAUC (`codesign`). See `-purpose` and `VERIFY OPERATION` in the OpenSSL verify manual page and the *Certificate Key Usage Attributes* section for more information.

[casync] section

The `casync` section contains casync-related settings. For more information about using casync support of RAUC, refer to *RAUC casync Support*.

`storepath` Allows to set the path to use as chunk store path for casync to a fixed one. This is useful if your chunk store is on a dedicated server and will be the same pool for each update you perform. By default, the chunk store path is derived from the location of the RAUC bundle you install.

`tmppath` Allows to set the path to use as temporary directory for casync. The temporary directory used by casync can be specified using the TMPDIR environment variable. It falls back to /var/tmp if unset. If `tmppath` is set then RAUC runs casync with TMPDIR sets to that path. By default, the temporary directory is left unset by RAUC and casync uses its internal default value `/var/tmp`.

[autoinstall] section

The auto-install feature allows to configure a path that will be checked upon RAUC service startup. If there is a bundle placed under this specific path, this bundle will be installed automatically without any further interaction.

This feature is useful for automatically updating the slot RAUC currently runs from, like for asymmetric redundancy setups where the update is always performed from a dedicated (recovery) slot.

path The full path of the bundle file to check for. If file at `path` exists, auto-install will be triggered.

[handlers] section

Handlers allow to customize RAUC by placing scripts in the system that RAUC can call for different purposes. All parameters expect pathnames to the script to be executed. Pathnames are either absolute or relative to the system.conf file location.

RAUC passes a set of environment variables to handler scripts. See details about using handlers in *Custom Handlers (Interface)*.

system-info This handler will be called when RAUC starts up, right after loading the system configuration file. It is used for obtaining further information about the individual system RAUC runs on. The handler script must print the information to standard output in form of key value pairs `KEY=value`. The following variables are supported:

RAUC_SYSTEM_SERIAL Serial number of the individual board

pre-install This handler will be called right before RAUC starts with the installation. This is after RAUC has verified and mounted the bundle, thus you can access bundle content.

post-install This handler will be called after a successful installation. The bundle is still mounted at this moment, thus you could access data in it if required.

bootloader-custom-backend This handler will be called to trigger the following actions:

- get the primary slot
- set the primary slot
- get the boot state
- set the boot state

if a custom bootloader backend is used. See *Custom* for more details.

Note: When using a full custom installation (see *[handler] section*) RAUC will not execute any system handler script.

[slot.<slot-class>.<idx>] section

Each slot is identified by a section starting with `slot.` followed by the slot class name, and a slot number. The *<slot-class>* name is used in the *update manifest* to target the correct set of slots. It must not contain any . (dots) as these are used as hierarchical separator.

device=</path/to/dev> The slot's device path. This one is mandatory.

type=<type> The type describing the slot. Currently supported `<type>` values are `raw`, `nand`, `ubivol`, `ubifs`, `ext4`, `vfat`. See table *Slot Type* for a more detailed list of these different types. Defaults to `raw` if none given.

bootname=<name> Registers the slot for being handled by the *bootselection interface* with the `<name>` specified. The value must be unique across all slots. Only slots without a `parent` entry can have a `bootname`. The actual meaning of the name provided depends on the bootloader implementation used.

parent=<slot> The `parent` entry is used to bind additional slots to a bootable root file system `<slot>`. Indirect parent references are discouraged, but supported for now. This is used together with the `bootname` to identify

the set of currently active slots, so that the inactive one can be selected as the update target. The parent slot is referenced using the form `<slot-class>.<idx>`.

allow-mounted=<true/false> Setting this entry `true` tells RAUC that the slot may be updated even if it is already mounted. Such a slot can be updated only by a custom install hook.

readonly=<true/false> Marks the slot as existing but not updatable. May be used for sanity checking or informative purpose. A `readonly` slot cannot be a target slot.

install-same=<true/false> If set to `false`, this will tell RAUC to skip writing slots that already have the same content as the one that should be installed. Having the 'same' content means that the hash value stored for the target slot and the hash value of the update image are equal. The default value is `true` here, meaning that no optimization will be done as this can be unexpected if RAUC is not the only one that potentially alters a slot's content.

This replaces the deprecated entries `ignore-checksum` and `force-install-same`.

resize=<true/false> If set to `true` this will tell RAUC to resize the filesystem after having written the image to this slot. This only has an effect when writing an ext4 file system to an ext4 slot, i.e. if the slot has''type=ext4'' set.

extra-mount-opts=<options> Allows to specify custom mount options that will be passed to the slots `mount` call as `-o` argument value.

10.2 Manifest

The manifest file located in a RAUC bundle describes the images packed in the bundle and their corresponding target slot class.

A valid RAUC manifest file must be named `manifest.raucm`.

```
[update]
compatible=FooCorp Super BarBazzer
version=2016.08-1

[bundle]
format=verity
verity-hash=3fcb193cb4fd475aa174efa1f1e979b2d649bf7f8224cc97f4413b5ee141a4e9
verity-salt=4b7b8657d03759d387f24fb7bb46891771e1b370fff38c70488e6381d6a10e49
verity-size=24576

[image.rootfs]
filename=rootfs.ext4
size=419430400
sha256=b14c1457dc10469418b4154fef29a90e1ffb4dddd308bf0f2456d436963ef5b3

[image.appfs]
filename=appfs.ext4
size=219430400
sha256=ecf4c031d01cb9bfa9aa5ecfce93efcf9149544bdbf91178d2c2d9d1d24076ca
```

[update] section

compatible A user-defined compatible string that must match the compatible string of the system the bundle should be installed on.

version A free version field that can be used to provide and track version information. No checks will be performed on this version by RAUC itself, although a handler can use this information to reject updates.

description A free-form description field that can be used to provide human-readable bundle information.

build A build id that would typically hold the build date or some build information provided by the bundle creation environment. This can help to determine the date and origin of the built bundle.

[bundle] section

format Either `plain` (default) or `verity`. This selects the *format* use when wrapping the payload during bundle creation.

verity-hash The dm-verity root hash over the bundle payload in hexadecimal. RAUC determines this value automatically, so it should be left unspecified when preparing a manifest for bundle creation.

verity-salt The dm-verity salt over the bundle payload in hexadecimal. RAUC determines this value automatically, so it should be left unspecified when preparing a manifest for bundle creation.

verity-size The size of the dm-verity hash tree. RAUC determines this value automatically, so it should be left unspecified when preparing a manifest for bundle creation.

[hooks] section

filename Hook script path name, relative to the bundle content.

hooks List of hooks enabled for this bundle. See *Install Hooks* for more details.

Valid items are: `install-check`

[handler] section

filename Handler script path name, relative to the bundle content. Used to fully replace default update process.

args Arguments to pass to the handler script, such as `args=--verbose`

[image.<slot-class>] section

filename Name of the image file (relative to bundle content). RAUC uses the file extension and the slot type to decide how to extract the image file content to the slot.

sha256 sha256 of image file. RAUC determines this value automatically when creating a bundle, thus it is not required to set this by hand.

size size of image file. RAUC determines this value automatically when creating a bundle, thus it is not required to set this by hand.

hooks List of per-slot hooks enabled for this image. See *Slot Hooks* for more details.

Valid items are: `pre-install`, `install`, `post-install`

10.3 Bundle Formats

RAUC currently supports two bundle formats (`plain` and `verity`) and additional formats could be added to support features such as encryption. Version 1.4 (released on 2020-06-20) and earlier only supported a single format which is now named `plain`, which should be used as long as compatibility to those versions is required.

The `verity` format was added to prepare for future use cases (such as network streaming and encryption), for better parallelization of installation with hash verification and to detect modification of the bundle during installation.

The bundle format is detected when reading a bundle and checked against the set of allowed formats configured in the `system.conf` (see *bundle-formats*).

10.3.1 plain Format

In this case, a bundle consists of:

- squashfs filesystem containing manifest and images
- detached CMS signature over the squashfs filesystem
- size of the CMS signature

With this format, the signature is checked in a full pass over the squashfs before mounting or accessing it. This makes it necessary to protect the bundle against modification by untrusted processes. To ensure exclusive access, RAUC takes ownership of the file (using chown) and uses file leases to detect other open file descriptors.

10.3.2 verity Format

In this case, a bundle consists of:

- squashfs filesystem containing manifest (without verity metadata) and images
- dm-verity hash tree over the squashfs filesystem
- CMS signature over an inline manifest (with verity metadata)
- size of the CMS signature

With this format, the manifest is contained in the CMS signature itself, making it accessible without first hashing the full squashfs. The manifest contains the additional metadata (*root hash, salt and size*) necessary to authenticate the hash tree and in turn each block of the squashfs filesystem.

During installation, the kernel's verity device mapper target is used on top of the loopback block device to authenticate each filesystem block as needed.

When using *rauc extract* (or other commands which need access to the squashfs except *install*), the squashfs is checked before accessing it by RAUC itself without using the kernel's device mapper target, as they are often used by normal users on their development hosts. It this case, the same mechanism for ensuring exclusive access as with plain bundles is used.

10.4 Slot Status

There is some slot specific metadata that are of interest for RAUC, e.g. a hash value of the slot's content (SHA-256 per default) that is matched against its counterpart of an image inside a bundle to decide if an update of the slot has to be performed or can be skipped. These slot metadata can be persisted in one of two ways: either in a slot status file stored on each slot containing a writable filesystem or in a central status file that lives on a persistent filesystem untouched by updates. The former is RAUC's default whereas the latter mechanism is enabled by making use of the optional key *statusfile* in the `system.conf` file. Both are formatted as INI-like key/value files where the slot information is grouped in a section named [slot] for the case of a per-slot file or in sections termed with the slot name (e.g. [slot.rootfs.1]) for the central status file:

```
[slot]
bundle.compatible=FooCorp Super BarBazzer
bundle.version=2016.08-1
bundle.description=Introduction of Galactic Feature XYZ
bundle.build=2016.08.1/imx6/20170324-7
status=ok
sha256=b14c1457dc10469418b4154fef29a90e1ffb4dddd308bf0f2456d436963ef5b3
size=419430400
```

(continues on next page)

(continued from previous page)

```
installed.timestamp=2017-03-27T09:51:13Z
installed.count=3
```

For a description of `sha256` and `size` keys see *this* part of the section *Manifest*. Having the slot's content's size allows to re-calculate the hash via `head -c <size> <slot-device> | sha256sum` or `dd bs=<size> count=1 if=<slot-device> | sha256sum`.

The properties `bundle.compatible`, `bundle.version`, `bundle.description` and `bundle.build` are copies of the respective manifest properties. More information can be found in this *subsection* of section *Manifest*.

RAUC also stores the point in time of installing the image to the slot in `installed.timestamp` as well as the number of updates so far in `installed.count`. Additionally RAUC tracks the point in time when a bootable slot is activated in `activated.timestamp` and the number of activations in `activated.count`, see section *Manually Switch to a Different Slot*. Comparing both timestamps is useful to decide if an installed slot has ever been activated or if its activation is still pending.

10.5 Command Line Tool

```
Usage:
  rauc [OPTION...] <COMMAND>

Options:
  -c, --conf=FILENAME               config file
  --cert=PEMFILE|PKCS11-URL         cert file or PKCS#11 URL
  --key=PEMFILE|PKCS11-URL          key file or PKCS#11 URL
  --keyring=PEMFILE                 keyring file
  --intermediate=PEMFILE            intermediate CA file name
  --mount=PATH                      mount prefix
  --override-boot-slot=BOOTNAME     override auto-detection of booted slot
  --handler-args=ARGS               extra handler arguments
  -d, --debug                       enable debug output
  --version                         display version
  -h, --help

List of rauc commands:
  bundle        Create a bundle
  resign        Resign an already signed bundle
  convert       Convert classic to casync bundle
  extract       Extract the bundle content
  install       Install a bundle
  info          Show file information
  service       Start RAUC service
  status        Show status
  write-slot    Write image to slot and bypass all update logic

Environment variables:
  RAUC_PKCS11_MODULE  Library filename for PKCS#11 module (signing only)
  RAUC_PKCS11_PIN     PIN to use for accessing PKCS#11 keys (signing only)
```

10.6 Custom Handlers (Interface)

Interaction between RAUC and custom handler shell scripts is done using shell variables.

RAUC_SYSTEM_CONFIG Path to the system configuration file (default path is `/etc/rauc/system.conf`)

RAUC_CURRENT_BOOTNAME Bootname of the slot the system is currently booted from

RAUC_BUNDLE_MOUNT_POINT Path to mounted update bundle, e.g. `/mnt/rauc/bundle`

RAUC_UPDATE_SOURCE A deprecated alias for `RAUC_BUNDLE_MOUNT_POINT`

RAUC_MOUNT_PREFIX Provides the path prefix that may be used for RAUC mount points

RAUC_SLOTS An iterator list to loop over all existing slots. Each item in the list is an integer referencing one of the slots. To get the slot parameters, you have to resolve the per-slot variables (suffixed with <N> placeholder for the respective slot number).

RAUC_TARGET_SLOTS An iterator list similar to `RAUC_SLOTS` but only containing slots that were selected as target slots by the RAUC target slot selection algorithm. You may use this list for safely installing images into these slots.

RAUC_SLOT_NAME_<N> The name of slot number <N>, e.g. `rootfs.0`

RAUC_SLOT_CLASS_<N> The class of slot number <N>, e.g. `rootfs`

RAUC_SLOT_TYPE_<N> The type of slot number <N>, e.g. `raw`

RAUC_SLOT_DEVICE_<N> The device path of slot number <N>, e.g. `/dev/sda1`

RAUC_SLOT_BOOTNAME_<N> The bootloader name of slot number <N>, e.g. `system0`

RAUC_SLOT_PARENT_<N> The name of slot number <N>, empty if none, otherwise name of parent slot

```
for i in $RAUC_TARGET_SLOTS; do
        eval RAUC_SLOT_DEVICE=\$RAUC_SLOT_DEVICE_${i}
        eval RAUC_IMAGE_NAME=\$RAUC_IMAGE_NAME_${i}
        eval RAUC_IMAGE_DIGEST=\$RAUC_IMAGE_DIGEST_${i}
done
```

10.7 Hooks (Interface)

10.7.1 Install Hooks Interface

The following environment variables will be passed to the hook executable:

RAUC_SYSTEM_COMPATIBLE The compatible value set in the system configuration file, e.g. `"My First Product"`

RAUC_SYSTEM_VARIANT The system's variant as obtained by the variant source (refer ref:*sec-variants*)

RAUC_MF_COMPATIBLE The compatible value provided by the current bundle, e.g. `"My Other Product"`

RAUC_MF_VERSION The value of the version field as provided by the current bundle, e.g. `"V1.2.1-2020-02-28"`

RAUC_MOUNT_PREFIX The global RAUC mount prefix path, e.g. `"/run/mount/rauc"`

10.7.2 Slot Hooks Interface

The following environment variables will be passed to the hook executable:

RAUC_SYSTEM_COMPATIBLE The compatible value set in the system configuration file, e.g. `"My Special Product"`

RAUC_SYSTEM_VARIANT The system's variant as obtained by the variant source (refer ref:*sec-variants*)

RAUC_SLOT_NAME The name of the currently installed slot, e.g `"rootfs.1"`.

RAUC_SLOT_STATE The state of the currently installed slot (will always be `inactive` for slots we install to)

RAUC_SLOT_CLASS The class of the currently installed slot, e.g. `"rootfs"`

RAUC_SLOT_TYPE The type of the currently installed slot, e.g. `"ext4"`

RAUC_SLOT_DEVICE The device path of the currently installed slot, e.g. `"/dev/mmcblk0p2"`

This equals the `device=` parameter set in the current slot's system.conf entry and represents the target device RAUC installs the update to. For an `install` hook, this is the device the hook executable should write to.

RAUC_SLOT_BOOTNAME For slots with a bootname (those that can be selected by the bootloader), the bootname of the currently installed slot, e.g. `"system1"` For slots with a parent, the parent's bootname is used. Note that in many cases, it's better to use the explicit `RAUC_SLOT_NAME` to select different behaviour in the hook, than to rely indirectly on the bootname.

RAUC_SLOT_PARENT If set, the parent of the currently installed slot, e.g. `"rootfs.1"`

RAUC_SLOT_MOUNT_POINT If available, the mount point of the currently installed slot, e.g. `"/run/mount/rauc/rootfs.1"`

For mountable slots, i.e. those with a file system type, RAUC will attempt to automatically mount the slot if a pre-install or post-install hook is given and provide the slot's current mount point under this env variable.

RAUC_IMAGE_NAME If set, the file name of the image currently to be installed, e.g. `"product-rootfs.img"`

RAUC_IMAGE_DIGEST If set, the digest of the image currently to be installed, e.g. `"e29364a81c542755fd5b2c2461cd12b0610b67ceacabce41c102bba4202f2b43"`

RAUC_IMAGE_CLASS If set, the target class of the image currently to be installed, e.g. `"rootfs"`

RAUC_MOUNT_PREFIX The global RAUC mount prefix path, e.g. `"/run/mount/rauc"`

10.8 D-Bus API

RAUC provides a D-Bus API that allows other applications to easily communicate with RAUC for installing new firmware.

de.pengutronix.rauc.Installer

10.8.1 Methods

InstallBundle (IN s source, IN a{sv} args);

Install (IN s source); (deprecated)

Info (IN s bundle, s compatible, s version);

Mark (IN s state, IN s slot_identifier, s slot_name, s message);

GetSlotStatus (a(sa{sv}) slot_status_array);

GetPrimary s primary);

10.8.2 Signals

Completed (i result);

10.8.3 Properties

Operation readable s

LastError readable s

Progress readable (isi)

Compatible readable s

Variant readable s

BootSlot readable s

10.8.4 Description

10.8.5 Method Details

The InstallBundle() Method

```
de.pengutronix.rauc.Installer.InstallBundle()
Install (IN  s source, IN a{sv} args);
```

Triggers the installation of a bundle. This method call is non-blocking. After completion, the *"Completed"* signal will be emitted.

IN s *source*: Path to bundle to be installed

IN a{sv} *args*: Arguments to pass to installation

Currently supported:

STRING 'ignore-compatible', VARIANT 'b' <true/false> Ignore the default compatible check for forcing installation of bundles on platforms that a compatible not matching the one of the bundle to be installed

The Install() Method

Note: This method is deprecated.

```
de.pengutronix.rauc.Installer.Install()
Install (IN  s source);
```

Triggers the installation of a bundle. This method call is non-blocking. After completion, the *"Completed"* signal will be emitted.

IN s *source*: Path to bundle to be installed

The Info() Method

```
de.pengutronix.rauc.Installer.Info()
Info (IN  s bundle, s compatible, s version);
```

Provides bundle info.

IN s *bundle*: Path to bundle information should be shown

s *compatible*: Compatible of bundle

s *version*: Version string of bundle

The Mark() Method

```
de.pengutronix.rauc.Installer.Mark()
Mark (IN  s state, IN  s slot_identifier, s slot_name, s message);
```

Keeps a slot bootable (state == "good"), makes it unbootable (state == "bad") or explicitly activates it for the next boot (state == "active").

IN s *state*: Operation to perform (one out of "good", "bad" or "active")

IN s *slot_identifier*: Can be "booted", "other" or <SLOT_NAME> (e.g. "rootfs.1")

s *slot_name*: Name of the slot which has ultimately been marked

s *message*: Message describing what has been done successfully (e.g. "activated slot rootfs.0")

The GetSlotStatus() Method

```
de.pengutronix.rauc.Installer.GetSlotStatus()
GetSlotStatus (a(sa{sv}) slot_status_array);
```

Access method to get all slots' status.

a(sa{sv}) *slot_status_array*: Array of (slotname, dict) tuples with each dictionary representing the status of the corresponding slot

The GetPrimary() Method

```
de.pengutronix.rauc.Installer.GetPrimary()
GetPrimary (s primary);
```

Get the current primary slot.

10.8.6 Signal Details

The "Completed" Signal

```
de.pengutronix.rauc.Installer::Completed
Completed (i result);
```

This signal is emitted when an installation completed, either successfully or with an error.

i *result*: return code (0 for success)

10.8.7 Property Details

The "Operation" Property

```
de.pengutronix.rauc.Installer:Operation
Operation  readable   s
```

Represents the current (global) operation RAUC performs. Possible values are `idle` or `installing`.

The "LastError" Property

```
de.pengutronix.rauc.Installer:LastError
LastError  readable   s
```

Holds the last message of the last error that occurred.

The "Progress" Property

```
de.pengutronix.rauc.Installer:Progress
Progress  readable   (isi)
```

Provides installation progress information in the form

(percentage, message, nesting depth)

Refer *Processing Progress Data* section.

The "Compatible" Property

```
de.pengutronix.rauc.Installer:Compatible
Compatible  readable   s
```

Represents the system's compatible. This can be used to check for usable bundles.

The "Variant" Property

```
de.pengutronix.rauc.Installer:Variant
Variant  readable   s
```

Represents the system's variant. This can be used to select parts of an bundle.

The "BootSlot" Property

```
de.pengutronix.rauc.Installer:BootSlot
BootSlot  readable   s
```

Contains the information RAUC uses to identify the booted slot. It is derived from the kernel command line. This can either be the slot name (e.g. `rauc.slot=rootfs.0`) or the root device path (e.g. `root=PARTUUID=0815`). If the `root=` kernel command line option is used, the symlink is resolved to the block device (e.g. `/dev/mmcblk0p1`).

10.9 RAUC's Basic Update Procedure

Performing an update using the default RAUC mechanism will work as follows:

1. Startup, read system configuration
2. Determine slot states
3. Verify bundle signature (reject if invalid)
4. Mount bundle (SquashFS)
5. Parse and verify manifest
6. Determine target install group
   A. Execute *pre install handler* (optional)
7. Verify bundle compatible against system compatible (reject if not matching)
8. Mark target slots as non-bootable for bootloader
9. Iterate over each image specified in the manifest
   A. Determine update handler (based on image and slot type)
   B. Try to mount slot and read slot status information
      a. Skip update if new image hash matches hash of installed one
   C. Perform slot update (image copy / mkfs+tar extract / . . .)
   D. Try to write slot status information
10. Mark target slots as new primary boot source for the bootloader
    A. Execute *post install* handler (optional)
11. Unmount bundle
12. Terminate successfully if no error occurred

10.10 Bootloader Interaction

RAUC comes with a generic interface for interacting with the bootloader. It handles *all* slots that have a *bootname* property set.

It provides two base functions:

1) Setting state 'good' or 'bad', reflected by API routine *r_boot_set_state()* and command line tool option *rauc status mark <good/bad>*
2) Marking a slot 'primary', reflected by API routine *r_boot_set_primary()* and command line tool option *rauc status mark-active*

The default flow of how they will be called during the installation of a new bundle (on Slot 'A') looks as follows:

The aim of setting state 'bad' is to disable a slot in a way that the bootloader will not select it for booting anymore. As shown above this is either the case before an installation to make the update atomic from the bootloader's perspective, or optionally after the installation and a reboot into the new system, when a service detects that the system is in an unusable state. This potentially allows falling back to a working system.

The aim of setting a slot 'primary' is to let the bootloader select this slot upon next reboot in case of having completed the installation successfully. An alternative to directly marking a slot primary after installation is to manually mark it primary at a later point in time, e.g. to let a complete set of devices change their software revision at the same time.

Setting the slot 'good' is relevant for the first boot but for all subsequent boots, too. In most cases, this interaction with the bootloader is required by the mechanism that enables fallback capability; rebooting a system one or several times without calling *rauc status mark-good* will let the bootloader boot an alternative system or abort boot operation (depending on configuration). Usually, bootloaders implement this fallback mechanism by some kind of counters they maintain and decrease upon each boot. In these cases *marking good* means resetting these counters.

A normal reboot of the system will look as follows:

Some bootloaders do not require explicitly setting state 'good' as they are able to differentiate between a POR and a watchdog reset, for example.

What the high-level functions described above actually do mainly depends on the underlying bootloader used and the capabilities it provides. Below is a short description about behavior of each bootloader interface currently implemented:

10.10.1 U-Boot

The U-Boot implementation assumes to have variables *BOOT_ORDER* and *BOOT_x_ATTEMPTS* handled by the bootloader scripting.

state bad Sets the *BOOT_x_ATTEMPTS* variable of the slot to *0* and removes it from the *BOOT_ORDER* list

state good Sets the *BOOT_x_ATTEMPTS* variable back to its default value (*3*).

primary Moves the slot from its current position in the list in *BOOT_ORDER* to the first place and sets *BOOT_x_ATTEMPTS* to its initial value (*3*). If BOOT_ORDER was unset before, it generates a new list of all slots known to RAUC with the one to activate at the first position.

10.10.2 Barebox

The barebox implementation assumes using barebox bootchooser.

state bad Sets both the *bootstate.systemX.priority* and *bootstate.systemX.remaining_attempts* to *0*.

state good Sets the *bootstate.systemX.remaining_attempts* to its default value (*3*).

primary Sets *bootstate.systemX.priority* to *20* and all other priorities that were non-zero before to *10*. It also sets *bootstate.systemX.remaining_attempts* to its initial value (*3*).

10.10.3 GRUB

state bad Sets slot *x_OK* to *0* and resets *x_TRY* to *0*.

state good Sets slot *x_OK* to *1* and resets *x_TRY* to *0*.

primary Sets slot *x_OK* to *1* and resets *x_TRY* to *0*. Sets *ORDER* to contain slot `x` as first element and all other after.

10.10.4 EFI

state bad Removes the slot from *BootOrder*

state good Prepends the slot to the *BootOrder* list. This behaves slightly different than the other implementations because we use *BootNext* for allowing setting primary with an initial fallback option. Setting state good is then used to persist this.

primary Sets the slot as *BootNext* by default. This will make the slot being booted upon next reboot only!

The behavior is different when `efi-use-bootnext` is set to `false`. Then this prepends the slot to the *BootOrder* list as described for 'state good'.

Note: EFI implementations differ in how they handle new or unbootable targets etc. It may also depend on the actual implementation if EFI variable writing is atomic or not. Thus make sure your EFI works as expected and required.

CHAPTER 11

Terminology

Update Controller This controls the update process and can be started on demand or run as a daemon.

Update Handler The handler performs the actual update installation. A default implementation is provided with the **update controller** and can be overridden in the **update manifest**.

Update Bundle The bundle is a single file containing an update. It consists of a squashfs with an appended cryptographic signature. It contains the **update manifest**, one or more images and optionally an **update handler**.

Update Manifest This contains information about update compatibility, image hashes and references the optional **handler**. It is either contained in a **bundle** or downloaded individually over the network.

Slot Slots are possible targets for (parts of) updates. Usually they are partitions on a SD/eMMC, UBI volumes on NAND/NOR flash or raw block devices. For filesystem slots, the **controller** stores status information in a file in that filesystem.

Slot Class All slots with the same purpose (such as rootfs, appfs) belong to the same **slot class**. Only one slot per class can be active at runtime.

Install Group If a system consists of more than only the root file system, additional slots are bound to one of the root file system slots. They form an **install group**. An update can be applied only to members of the same group.

System Configuration This configures the **controller** and contains compatibility information and slot definitions. For now, this file is shipped as part of the root filesystem.

Boot Chooser The bootloader component that determines which slot to boot from.

Recovery System A non-updatable initial (factory default) system, capable of running the update service to recover the system if all other slots are damaged.

CHAPTER 12

Contributing

Thank you for thinking about contributing to RAUC! Some different backgrounds and use-cases are essential for making RAUC work well for all users.

The following should help you with submitting your changes, but don't let these guidelines keep you from opening a pull request. If in doubt, we'd prefer to see the code earlier as a work-in-progress PR and help you with the submission process.

12.1 Workflow

- Changes should be submitted via a GitHub pull request.
- Try to limit each commit to a single conceptual change.
- Add a signed-of-by line to your commits according to the *Developer's Certificate of Origin* (see below).
- Check that the tests still work before submitting the pull request. Also check the CI's feedback on the pull request after submission.
- When adding new features, please also add the corresponding documentation and test code.
- If your change affects backward compatibility, describe the necessary changes in the commit message and update the examples where needed.

12.2 Code

- Basically follow the Linux kernel coding style

12.3 Documentation

- Use semantic linefeeds in .rst files.

12.4 Developer's Certificate of Origin

RAUC uses the Developer's Certificate of Origin 1.1 with the same process as used for the Linux kernel:

> Developer's Certificate of Origin 1.1
>
> By making a contribution to this project, I certify that:
>
> (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
>
> (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
>
> (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
>
> (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Then you just add a line (using `git commit -s`) saying:

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

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

CHAPTER 13

Changes in RAUC

13.1 Release 1.5.1 (released Jan 22, 2021)

Bug fixes

- Fix building with kernel headers < 4.14. (by Fabrice Fontaine)
- Fix manifest generation for casync bundles.
- Fix too strict payload size check which triggered on casync bundles generated by versions up to 1.4.
- Restore compatibility with glib 2.50.

Testing

- Switch from Travis-CI to GitHub actions.
- Add test builds on Ubuntu 16.04, 18.04 and 20.04 to catch build problems with older environments.

Contributions from: Enrico Jörns, Fabrice Fontaine, Jan Lübbe

13.2 Release 1.5 (released Dec 14, 2020)

Note: This version introduces the new `verity` bundle format (the old format is now called `plain`). The `verity` format was added to prepare for future use cases (such as network streaming and encryption), for better parallelization of installation with hash verification and to detect modification of the bundle during installation (CVE-2020-25860). The bundle format is detected when reading a bundle and checked against the set of allowed formats configured in the system.conf (see *Bundle Formats*).

As the old `plain` format does not offer protection against modification during the installation process, RAUC now takes ownership of the bundle file, removes write permissions and checks for existing open file descriptors. This is

intended as a mitigation to protect against a compromised update service running as a non-root user, which would otherwise be able to modify the bundle between signature check and actual bundle installation.

See *Bundle Format Migration* for more details on how to switch to the `verity` format.

Enhancements

- Add support for the `verity` bundle format. See the *reference for details*.
- Support resolving the *root=PARTLABEL=xxx* kernel command line option. (by Gaël PORTAY)
- Disable the unneccessary SMIMECapabilities information in the bundle signature, saving ~100 bytes.
- Remove redundant checksum verification for source images during installation. The RAUC bundle is already verified at this point, so there is no need to verify the checksum of each file individually. (by Bastian Krause)

Security

- Take ownership of bundle files if they are not owned by root and remove write permissions. Then check that no writable file descriptors are open for the bundle file (using the `F_SETLEASE` fcntl). This fixes CVE-2020-25860. See the advisory for more details: https://github.com/rauc/rauc/security/advisories/GHSA-cgf3-h62j-w9vv

Note: The https://github.com/rauc/rauc-1.5-integration repository contains examples to simplify integrating the RAUC update into existing projects. You can subscribe to https://github.com/rauc/rauc-1.5-integration/issues/1 to receive notifications of important updates to this repository and of integration into the upstream build systems.

Bug fixes

- Fix install handler selection for *.img files for boot-* slots when used with casync. (by Martin Schwan)
- Fix checking for unknown keys in the slot configuration.
- Fix some corner cases related to stopping the D-Bus daemon.
- Propagate error if unable to save manifest. (by Stefan Wahren)
- Apply *–handler-args* only during installation (and not during bundle creation).

Testing

- Ship *test/minimal-test.conf* to fix testing when running as root. (by Uwe Kleine-König)
- Increase usage of g_autofree/g_autoptr in the test suite.

Code

- Remove unused code for signed manifests (outside of a bundle).
- Add G_GNUC_WARN_UNUSED_RESULT to many functions.

Documentation

- Fix multiple smaller errors. (by Christoph Steiger, Christopher Obbard and Michael Heimpold)
- Improve documentation related to u-boot scripting and environment storage.

Contributions from: Bastian Krause, Christoph Steiger, Christopher Obbard, Enrico Jörns, Gaël PORTAY, Jan Lübbe, Martin Schwan, Michael Heimpold, Stefan Wahren, Uwe Kleine-König

13.3 Release 1.4 (released Jul 20, 2020)

Note: Slots with both a `parent=` and a `bootname=` entry are now rejected when parsing the system configuration. While the intention was to have either a bootname or a parent link, this was not enforced in previous versions. Move the bootname to the parent slot when updating to RAUC 1.4.

It is now recommended to explicitly select either per-slot or global configuration file in the system config using `statusfile=<path>/per-slot`. If a central storage location is available, global status file should be preferred.

Enhancements

- Added support for custom boot selection scripts/binaries. This allows handling special cases where none of the standard bootloaders is available for switching the redundant slots. (by Christian Bräuner Sørensen, *docs* by Andreas Schmidt)
- Changed ext4 filesystem creation options to always use 256 byte inodes. Without it, mkfs.ext4 will default to 128 byte inodes on filesystems smaller than 512MiB. This avoids the "ext4 filesystem being mounted at /foo supports timestamps until 2038" message on newer kernels.
- Added new slot type `boot-gpt-switch` to support atomic updating of boot partitions in the GPT. This is useful if the firmware does not support atomic bootloader updates by itself. See *here* for details.

Bug fixes

- Improve parent and bootname consistency checks when loading the system config. (by Dan Callaghan)
- Fix and improve installation log output for the –disable-service configuration.
- Clean up incomplete bundles on creation errors consistently for extract/resign/convert and doesn't remove pre-existing files anymore.
- Fix minor memory leaks.

Testing

- Added tests for UBIFS and NAND slot types via nandsim in qemu.
- Added CI testing of the –disable-service configure option.
- Added test cases for some CLI subcommands.

Code

- Clarified licensing of the D-Bus API file. (by Michael Heimpold)

Documentation

- Manual pages have been updated with new options. (by Michael Heimpold)
- Improved documentation around central and per-slot status files.
- Improved images and various text sections.

Contributions from: Andreas Schmidt, Bastian Krause, Christian Bräuner Sørensen, Dan Callaghan, Enrico Jörns, Jan Lübbe, Michael Heimpold, Tobias Junghans, Uwe Kleine-König

13.4 Release 1.3 (released Apr 23, 2020)

Enhancements

- Added a new D-Bus method (InstallBundle) which supports optional parameters ("ignore-compatible" for now).
- Added support for X.509 key usage attributes (code signing and others).
- Added a `check-crl` configuration option to require Certificate Revocation List (CRL) checking during installation. If the keyring already contains a CRL, but checking is not enabled, a warning will be printed.
- Support updating of already mounted slots via a custom install hook when enabled with "allow-mounted=true" in the system configuration. This can be useful for updating bootloaders in a boot partition (for example on the Raspberry Pi or BeagleBone). (by Martin Hundebøll and Rasmus Villemoes)
- Added the `--mksquashfs-args` option for bundle creation. This can be used to configure the details of the squashfs compression. (by Louis des Landes)
- Added the `--casync-args` option for the `rauc convert` command. This can be used to configure the details of the casync conversion. (by Christopher Obbard)
- Added support for installing UBIFS images via casync (depends on the casync PR https://github.com/systemd/casync/pull/227). (by Ulrich Ölmann)
- Enabled usage of `--no-verify` with `rauc resign`. This can be useful for resigning of bundles signed with expired certificates.
- Exposed the `RAUC_BUNDLE_MOUNT_POINT` environment variable to hook scripts. This also deprecates the old name `RAUC_UPDATE_SOURCE` for this value in handler scripts. (by Rasmus Villemoes)
- Reduced size of the installed `rauc` binary. This was done by using `--gc-sections` and adding a configure switch to disable the `bundle`, `resign` and `convert` commands. (by Rasmus Villemoes)
- Added support for explicitly telling RAUC that all slots are inactive on the kernel command line (`rauc.external`). This is useful for using RAUC in a factory installer. (by Marco Felsch)
- Improved layout of the `rauc status` output.

Bug fixes

- Fixed SD/eMMC detection when using /dev/disk/by-path/ symlinks. (by Marco Felsch)
- Fixed handling of HTTP Content-Encoding: gzip. (by Jan Kundrát)
- Fixed reporting of errors during bundle verification. This solves a `rauc-ERROR **: Not enough substeps: check_bundle` abort. (by Rouven Czerwinski)
- Fixed handling of surrounding whitespace in the system variant by removing it. A warning is printed in this case.

- Fixed the RAUC D-Bus interface introspection file name to be consistent with the interface name. (by Michael Tretter)

Testing

- Switched testing environment from user-mode-linux (UML) to QEMU. This allows us to use our own kernel configuration and avoids the (unusual) dependency.
- Reenabled support for coverity, as they have added support for GCC 8.
- Added some more tests in several areas.

Code

- Removed support for OpenSSL versions < 1.1.1. OpenSSL versions 1.0.2 and 1.1.0 are no longer supported by the OpenSSL project: https://www.openssl.org/policies/releasestrat.html
- Improved support for large bundles on 32 bit systems, but some work remains to be done.
- Disabled automatic `-Werror` and `-O0` when building from a git repository. This caused confusion in several cases.
- Updated uncrustify and enabled some additional formatting rules.
- Reduced redundant prefixes in error messages.
- Removed unused verification functions left over from the old network mode.
- Removed minor memory leaks.

Documentation

- Clarified documentation about hooks and handlers (and the available environment variables).
- Fixed minor typos and inconsistencies.

Contributions from: Arnaud Rebillout, Christopher Obbard, Enrico Jörns, Jan Kundrát, Jan Lübbe, Louis des Landes, Marco Felsch, Martin Hundebøll, Michael Heimpold, Michael Tretter, Rasmus Villemoes, Rouven Czerwinski, Trent Piepho, Ulrich Ölmann

13.5 Release 1.2 (released Oct 27, 2019)

Enhancements

- Added `--signing-keyring` argument to specify a distinct keyring for post-signing verification. This allows for example to use `rauc resign` with certs not verifying against the original keyring.
- Output of 'rauc status' is now grouped by slot groups to make it easier to identify the redundancy setup. Previously, the present slots were printed in a random order which was confusing, especially when having more than three or four slots.
- Use pkg-config to obtain valid D-Bus install directories and clean up D-Bus directory handling. This adds libdbus-1-dev as new build dependency. (by Michael Heimpold)
- Moved various checks that could be performed before actually starting the installation out of the atomic update region. This allows RAUC to fail earlier without leaving behind a disabled slot group with incomplete contents.

- Added optional `--progress` argument to `rauc install` that enables a basic text progress bar instead of the default line-by-line log.
- Added `tmppath` to casync system config options to allow setting TMPDIR for casync. (by Gaël PORTAY)
- Slot skipping was deactivated by default as it turned out to be unexpected behaviour for many users. The corresponding setting was renamed to 'install-same=' ('force-install-same' will remain valid, too). The means skipping writing for slots whose current and intended slot hashes are equal must now be enabled explicitly. This optimization is mainly useful for use-cases with a read-only rootfs.
- Added new slot type `boot-mbr-switch` to support atomic updating of boot partitions in the MBR. (by Thomas Hämmerle) See *here* for details.

Bug fixes

- Fixed detection of whether the bundle path is located in input directory for a corner case.
- Fixed off-by-one error in printing the remaining attempts counter in the uboot.sh contrib script (by Ellie Reeves)
- Fixed detection of mount points disappearing during the service's runtime.
- Added missing entry of 'service' subcommand to RAUC help text (if compiled with service support).
- Fixed inappropriate resetting of BOOT_ACK flag in eMMC extCSD register handling which could have prevented proper booting on some SoCs. (by Stephan Michaelsen)
- Fixed leaking GDataInputStreams in boot selection and install handling that led to steadily increasing number of open file descriptors in some scenarios until exceeding system limits and leading to 'Too many open files' errors. This was only problematic when installing many times without rebooting.
- Fixed 'uninitialized local' bugs in update_handler and config_file module. (by Gaël PORTAY)
- PKCS#11 handling now does not silently accept missing (empty) PINs anymore, but allows interactive prompt for entering it.
- Fixed bundle detection on big endian systems.
- Fixed size mismatches in printf formatter and struct packing on ARM32.

Testing

- Fix checks that depended on implicit assumptions regarding the GHashTable behaviour that are not valid anymore for newer glib versions.
- Added notes on required tools for unit testing and added check for grub-editenv being present.
- Travis now also runs cross-compilation tests for platforms armhf, i386, arm64, armel to allow early detection of cross-compilation issues with endianness, 32 vs. 64 bit, etc.

Code

- Reworked subprocess call logging for debugging and added remaining missing log output to users of r_subprocess_new().
- Refactored slot handling code in new 'slot.c' module to be used for both install and status information handling.
- Added qdbusxml2cpp annotations to rauc-installer.xml for interface class generation. (by Tobias Junghans)
- Removed the deprecated 'network mode'. Note that this does not affect RAUC's bundle network capabilities (casync, etc.).

- Fixed clang compilation warnings (unused variable, printf formatter, non-obvious invert statements).
- Various code cleanups, structural simplifications

Documentation

- Added hints for creating `/dev/data` symlink to mount the right data partition in dual data partition setups. (by Fabian Knapp)
- Extended manpage to cover 'rauc status' subcommands. (by Michael Heimpold)
- Fixed various typos.

Contributions from: Bastian Krause, Ellie Reeves, Enrico Jörns, Fabian Knapp, Gaël PORTAY, Jan Lübbe, Leif Middelschulte, Michael Heimpold , Stephan Michaelsen , Thomas Hämmerle, Thorsten Scherer, Tobias Junghans, Uwe Kleine-König

13.6 Release 1.1 (released Jun 5, 2019)

Enhancements

- Check that we do not generate a bundle inside a source directory
- Added full GRUB2 support, including status and primary slot readback (by Vitaly Ogoltsov and Beralt Meppelink)
- Allow passing a slot's name via commandline instead of it's bootname
- Show each slot's name in `Booted from` line of `rauc status` to simplify identification
- Add `resize` option for ext4 slots to let RAUC run resize2fs on an ext4 slot after copying the image.
- Allow dumping the signer certificate (`--dump-cert`) without verification
- Allow specifying a keyring directory with multiple files to support non-conflicting installations of certificates from different packages (by Evan Edstrom)
- Add a bootloader option `efi-use-bootnext` (only valid when bootloader is 'efi') to disable usage of BootNext for marking slots primary.
- Support setting a system variant in the `system-info` handler via `RAUC_SYSTEM_VARIANT`
- D-Bus "mountpoint" property now also exports external mount point
- Made slot state, compatible and variant available as environment variables for slot hooks
- Made system variant variable available as an environment variable for bundle hooks

Bug fixes

- Fix memory leaks in D-Bus notification callbacks (by Michael Heimpold)
- Fix memory leaks in resolve_bundle_path (by Michael Heimpold)
- Do not print misleading status dump when calling `mark-*` subcommands
- Avoid mmap'ing potentially huge files (by Rasmus Villemoes)
- Fix and cleanup checksum verification and handling (by Rasmus Villemoes)
- Avoid assertion error caused by unconditional slot status hash table freeing

- Make a-month-from-now validity check in signature verification more robust (by Rasmus Villemoes)

Testing

- Enable lgtm analysis for tests
- Restructure signature tests with set_up and tear_down (by Evan Edstrom)
- Move from gcc-6 to gcc-7
- Build environment fixes and workarounds

Code

- A failure in calling barebox_state bootchooser implementation should be propagated
- Update to latest `git-version-gen` upstream version
- Tail-call real rauc suprocess in `rauc-service.sh` (by Angus Lees)
- Consistently return newly-allocated objects in `resolve_path()`
- Enforce space between `if` and `(` via uncrustify

Documentation

- Added an initial version of a man page (by Michael Heimpold)
- Extended D-Bus API documentation
- Improve description of how RAUC detects the booted slot
- Added lgtm badge
- Add hints on library dependencies
- Clarifications on how to build and install RAUC
- Add note on basic RAUC buildroot support
- Clarification on usage of RAUC on host and target side
- Clarified documentation of 'use-bundle-signing-time' option (by Michael Heimpold)
- Typos fixed

Contributions from: Angus Lees, Arnaud Rebillout, Beralt Meppelink, Enrico Jörns, Evan Edstrom, Ian Abbott, Jan Lübbe, Michael Heimpold, Rasmus Villemoes, Ulrich Ölmann, Vitaly Ogoltsov

13.7 Release 1.0 (released Dec 20, 2018)

Enhancements

- Support OpenSSL 1.1
- Use OPENSSL_config() instead of OPENSSL_no_config()
- Handle curl_global_init() return code

Bug fixes

- Fix error handling when resolving the backing file for a loop device
- Fix error reporting when no primary slot is found with u-boot (by Matthias Bolte)
- Fix memory leaks when parsing handler output
- Fix compiler error when building with –disable-network
- Handle fatal errors during curl or openssl initialization
- Fix boot selection handling for asymmetric update setups
- Fix default variant string in case of failure when obtaining
- Fix return codes when giving excess arguments to CLI functions
- Let 'rauc service' return exit code != 0 in case of failure
- Print 'rauc service' user error output with g_printerr()
- Fix showing primary slot (obtained via D-Bus) in 'rauc status'
- Fix showing inverted boot-status (obtained via D-Bus) in 'rauc status'
- Minor output and error handling fixes and enhancements

Testing

- Fake entropy in uml tests to fix and speed up testing
- Fix creating and submitting coverity report data
- Migrate to using Docker images for testing
- Changed coverage service from coveralls to codecov.io
- Switch to uncrustify 0.68.1

Documentation

- Provided slot configuration *examples* for common scenarios
- Fixes and enhancements of README.rst to match current state
- Add sphinx DTS lexer for fixing and improving dts example code parsing

Contributions from: Ahmad Fatoum, Enrico Jörns, Jan Lübbe, Matthias Bolte

13.8 Release 1.0-rc1 (released Oct 12, 2018)

Enhancements

- Bundle creation
  - Add support for passing Keys/Certificates stored on PKCS#11 tokens (e.g. for using a smart card or HSM). See *PKCS#11 Support* for details.
  - Print a warning during signing if a certificate in the chain will expire within one month
  - If keyring is given during bundle creation, automatically verify bundle signature and trust chain

- Configuration (see the reference for the *[system]*, *[keyring]* and *[slot.*.*]* sections for details)
  - Add `extra-mount-opts` argument to slot config to allow passing custom options to `mount` calls (such as user_xattr or seclabel)
  - Implement support for `readonly` slots that are part of the slot description but should never be written by RAUC
  - Add option `use-bundle-signing-time` to use signing time for verification instead of the current time
  - Introduce `max-bundle-download-size` config setting (by Michael Heimpold)
  - Rename confusing `ignore-checksum` flag to `force-install-same` (old remains valid of course) (by Jan Remmet)
  - Add strict parsing of config files as we do for manifests already. This will reject configs with invalid keys, groups, etc. to prevent unintentional behavior
- Installation
  - Remove strict requirement of using `.raucb` file extension, although it is still recommended
  - Export RAUC slot type to handlers and hooks (by Rasmus Villemoes)
  - Add `*.squashfs` to `raw` slot handling (by Emmanuel Roullit)
  - Add checking of RAUC bundle identifier (squashfs identifier)
  - `*.img` files can now be installed to `ext4`, `ubifs` or `vfat` slots (by Michael Heimpold)
  - Warn if downloaded bundle could not be deleted
- Expose system information (variant, compatible, booted slot) over D-Bus (by Jan Remmet)
- The `rauc status` command line call now only uses the D-Bus API (when enabled) to obtain status information instead of loading configuration and performing operations itself. This finalizes the clear separations between client and service and also allows calling the command line client without requiring any configuration.
- Add debug log domain `rauc-subprocess` for printing RAUC subprocess invocations. This can be activated by setting the environment variable `G_MESSAGES_DEBUG=rauc-subprocess`. See *Debugging RAUC* for details.
- Enhancement of many debug and error messages to be more precise and helpful
- Let U-Boot boot selection handler remove slot from `BOOT_ORDER` when marking it bad
- Implemented obtaining state and primary information for U-Boot boot selection interface (by Timothy Lee)
- Also show certificate validity times when the certificate chain is displayed
- Added a simple CGI as an example on how to code against the D-Bus API in RAUC contrib/ folder. (by Bastian Stender)

Bug fixes

- Bootchooser EFI handler error messages and segfault fixed (by Arnaud Rebillout)
- Fix preserving of primary errors while printing follow-up errors in update_handlers (by Rasmus Villemoes)
- Make not finding (all) appropriate target slots a fatal error again
- Prevent non-installation operations from touching the installation progress information (by Bastian Stender)
- Call `fsync()` when writing raw images to assure content is fully written to disk before exiting (by Jim Brennan)

- Fix casync store initialization for extraction without seeds (by Arnaud Rebillout)
- Fix slot status path generation for external mounts (by Vyacheslav Yurkov)
- Do not try to mount already mounted slots when loading slot status information from per-slot file
- Fix invalid return value in case of failed `mark_active()`
- Fix bootname detection for missing `root=` command line parameter
- Fix passing intermediate certificates via command line which got broken by a faulty input check (by Marcel Hamer)
- Preserve original uid/gid during extraction to be independent of the running system. This was only problematic if the name to ID mapping changed with an update. Note that this requires to enable `CONFIG_FEATURE_TAR_LONG_OPTIONS` when using busybox tar.
- Block device paths are now opened with `O_EXCL` to ensure exclusive access
- Fix handling for `file://` URIs
- Build-fix workaround for ancient (< 3.4) kernels (by Yann E. MORIN)
- Various internal error handling fixes (by Ulrich Ölmann, Bastian Stender)
- Several memory leak fixes

Testing

- Abort on `g_critical()` to detect issues early
- Extended and restructured testing for barebox and u-boot boot selection handling
- Basic `rauc convert` (casync) testing
- Switch to Travis xenial environment
- Make diffs created by uncrustify fatal to enforce coding style
- Fix hanging rauc.t in case of failed tests for fixing sharness cleanup function handling
- Run sharness (rauc.t) tests with verbose output
- Show make-check log on error

Code

- Add GError handling to download functions
- Prepare support for tracing log level
- Start more detailed annotation of function parameter direction and transfer
- Simplified return handling as result of cleanup helper rework
- Treewide introduction of Glib automatic cleanup helpers. Increases minimum required GLib version to 2.45.8 (by Philipp Zabel)
- Prepare deprecation of RAUC ancient non-bundle 'network mode'

Documentation

- Add a *Debugging RAUC* chapter on how to debug RAUC
- Add a *Bootloader Interaction* section describing the boot selection layer and the special handling for the supported bootloaders
- Add hint on how to run RAUC without D-Bus to FAQ
- Document *Required Host Tools* and *Required Target Tools*
- Tons of typo fixes, minor enhancements, clarifications, example fixes, etc.

Contributions from: Alexander Dahl, Arnaud Rebillout, Bastian Stender, Emmanuel Roullit, Enrico Jörns, Jan Lübbe, Jan Remmet, Jim Brennan, Marcel Hamer, Michael Heimpold, Philip Downer, Philipp Zabel, Rasmus Villemoes, Thomas Petazzoni, Timothy Lee, Ulrich Ölmann, Vyacheslav Yurkov, Yann E. MORIN

13.9 Release 0.4 (released Apr 9, 2018)

Enhancements

- Add `barebox-statename` key to `[system]` section of system.conf in order to allow using non-default names for barebox state
- Support atomic bootloader updates for eMMCs. The newly introduced slot type `boot-emmc` will tell RAUC to handle bootloader updates on eMMC by using the `mmcblkXboot0/-boot1` partitions and the EXT_CSD registers for alternating updates.
- Support writing `*.vfat` images to vfat slots
- Add basic support for streaming bundles using casync tool. Using the casync tool allows streaming bundle updates chunk-wise over http/https/sftp etc. By using the source slot as a seed for the reproducible casync chunking algorithm, the actual chunks to download get reduced to only those that differ from the original system.
  - Add `rauc convert` command to convert conventional bundles to casync bundle and chunk store
  - Extend update handler to handle `.caibx` and `.caidx` suffix image types in bundle
- Added `--detailed` argument to `rauc status` to obtain newly added slot status information
- Added D-Bus Methods `GetSlotStatus` to obtain collected status of all slots
- Extended information stored in slot status files (installed bundle info, installation and activation timestamps and counters)
- Optionally use a central status file located in a storage location not touched during RAUC updates instead of per-slot files (enabled by setting `statusfile` key in `[system]` section of `system.conf`).
- Add `write-slot` command to write images directly to defined slots (for use during development)

Bug fixes

- Fix documentation out-of-tree builds
- Fixed packaging for dbus wrapper script rauc-service.sh
- Some double-free and error handling fixes

Testing

- Create uncrustify report during Travis run

Code

- Unified hash table iteration and variable usage
- Add uncrustify code style configuration checker script to gain consistent coding style. Committed changes revealed by initial run.

Documentation

- Updated and extended D-Bus interface documentation
- Added documentation for newly added features (casync, central slot status, etc.)
- Fixed and extended Yocto (meta-rauc) integration documentation
- Add link to IRC/Matrix channel
- Some minor spelling errors fixed

13.10 Release 0.3 (released Feb 1, 2018)

Enhancements

- Added support for intermediate certificates, improved bundle resigning and certificate information for hooks. This makes it easier to use a multi-level PKI with separate intermediate certificates for development and releases. See *Resigning Bundles* for details.
- Added support for image variants, which allow creating a single bundle which supports multiple hardware variants by selecting the matching image from a set contained in the bundle. See *Handling Board Variants With a Single Bundle* for details.
- Added support for redundant booting by using EFI boot entries directly. See *EFI* for details.
- Added boot information to `rauc status`
- Added `rauc extract` command to extract bundles
- Support detection of the booted slot by using the `UUID=` and `PARTUUID=` kernel options.
- Improved the status and error output
- Improved internal error cause propagation

Bug fixes

- Fixed boot slot detection for `root=<symlink>` boot parameters (such as `root=/dev/disk/by-path/pci-0000:00:17.0-ata-1-part1`)
- Removed redundant image checksum verification during installation.

Testing

- Improve robustness and test coverage
- Use gcc-7 for testing

Documentation

- Added documentation for
  - intermediate certificates
  - re-signing bundles
  - image variants
  - UEFI support
- Minor fixes and clarifications

13.11 Release 0.2 (released Nov 7, 2017)

Enhancements

- Added `--override-boot-slot` argument to force booted slot
- Display installation progress and error cause in CLI
- Allow installing uncompressed tar balls
- Error reporting for network handling and fail on HTTP errors
- Added `--keyring` command line argument
- Added `activate-installed` key and handling for `system.conf` that allows installing updates without immediately switching boot partitions.
- Extended `rauc status mark-{good,bad}` with an optional slot identifier argument
- Added subcommand `rauc status mark-active` to explicitly activate slots
- New D-Bus method `mark` introduced that allows slot activation via D-Bus
- Added `tar` archive update handler for `vfat` slots
- Introduced `rauc resign` command that allows to exchange RAUC signature without modifying bundle content
- Display signature verification trust chain in output of `rauc info`. Also generate and display SPKI hash for each certificate
- Added `--dump-cert` argument to `rauc info` to allow displaying signer certificate info

Documentation

- Added docs/, CHANGES and README to tarball
- Added and reworked a bunch of documentation chapters
- Help text for `rauc bundle` fixed

- Added short summary for command help

Bug fixes

- Flush D-Bus interface to not drop property updates
- Set proper PATH when starting service on non-systemd systems
- Include config.h on top of each file to fix largefile support and more
- Let CLI properly fail on excess arguments provided
- Do not disable bundle checking for `rauc info --no-verify`
- Properly clean up mount points after failures
- Abort on inconsistent slot parent configuration
- Misc memory leak fixes
- Fixes in error handling and debug printout
- Some code cleanups

Testing

- Miscellaneous cleanups, fixes and refactoring
- Add tests for installation via D-Bus
- Let Travis build documentation with treating warnings as errors
- Allow skipping sharness tests requiring service enabled
- Explicitly install dbus-x11 package to fix Travis builds
- Fix coveralls builds by using `--upgrade` during `pip install cpp-coveralls`
- Use gcc-6 for testing

13.12 Release 0.1.1 (released May 11, 2017)

Enhancements

- systemd service: allow systemd to manage and cleanup RAUCs mount directory

Documentation

- Added contribution guideline
- Added CHANGES file
- Converted README.md to README.rst
- Added RAUC logo
- Several typos fixed
- Updated documentation for mainline PTXdist recipes

Bug fixes

- Fix signature verification with OpenSSL 1.1.x by adding missing binary flag
- Fix typo in json status output formatter (“mountpint” -> “mountpoint”)
- Fixed packaging of systemd service files by removing generated service files from distribution
- src/context: initialize datainstream to NULL
- Added missing git-version-gen script to automake distribution which made autoreconf runs on release packages fail
- Fixed D-Bus activation of RAUC service for non-systemd systems

13.13 Release 0.1 (released Feb 24, 2017)

This is the initial release of RAUC.

- search
- genindex

CHAPTER 14

The Need for Updating

Updating an embedded system is always a critical step during the life cycle of an embedded hardware product. Updates are important to either fix system bugs, solve security problems or simply for adding new features to a platform.

As embedded hardware often is placed in locations that make it difficult or costly to gain access to the board itself, an update must be performed unattended; for example either by connecting a special USB stick or via some network roll-out strategy.

Updating an embedded system is risky; an update might be incompatible, a procedure crashes, the underlying storage fails with a write error, or someone accidentally switches the power off, etc. All this may occur but should not lead to having an unbootable hardware at the end.

Another point besides safe upgrades are security considerations. You would like to prevent that someone unauthorized is able to load modified firmware onto the system.

CHAPTER 15

What is RAUC?

RAUC is a lightweight update client that runs on your embedded device and reliably controls the procedure of updating your device with a new firmware revision. RAUC is also the tool on your host system that lets you create, inspect and modify update artifacts for your device.

The decision to design was made after having worked on custom update solutions for different projects again and again while always facing different issues and unexpected quirks and pitfalls that were not taken into consideration before.

Thus, the aim of RAUC is to provide a well-tested, solid and generic base for the different custom requirements and restrictions an update concept for a specific platform must deal with.

When designing the RAUC update tool, all of these requirements were taken into consideration. In the following, we provide a short overview of basic concepts, principles and solutions RAUC provides for updating an embedded system.

CHAPTER 16

And What Not?

RAUC is NOT a full-blown updating application or GUI. It provides a CLI for testing but is mainly designed to allow seamless integration into your individual Applications and Infrastructure by providing a D-Bus interface.

RAUC can NOT replace your bootloader who is responsible for selecting the appropriate target to boot, but it provides a well-defined interface to incorporate with all common bootloaders.

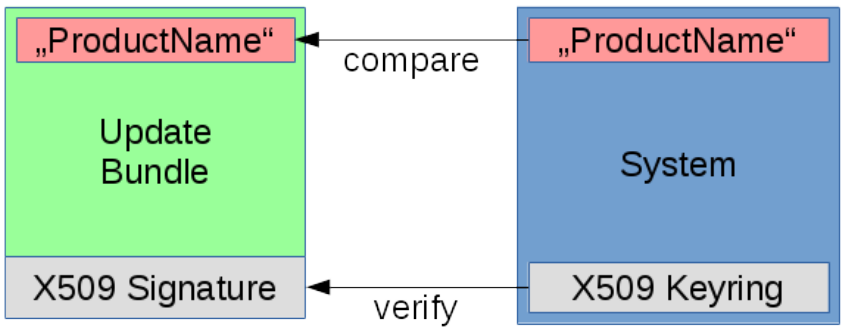
RAUC does NOT intend to be a deployment server. On your host side, it only creates the update artifacts. You may want to have a look at rauc-hawkbit for interfacing with the hawkBit deployment server.

And finally, factory bring up of your device, i.e. initial partitioning etc. is also out of scope for an update tool like RAUC. While you may use it for initially filling your slot contents during factory bring up, the partitioning or volume creation must be made manually or by a separate factory bring up script.

CHAPTER 17

Key Features of RAUC

- **Fail-Safe & Atomic**:
  - An update may be interrupted at any point without breaking the running system.
  - Update compatibility check
  - Mark boots as successful / failed
- **Cryptographic signing and verification** of updates using OpenSSL (signatures based on x.509 certificates)



  - Keys and certificates on **PKCS#11 tokens** (HSMs) are supported
- **Flexible and customizable** redundancy/storage setup
  - **Symmetric** setup (Root-FS A & B)
  - **Asymmetric** setup (recovery & normal)
  - Application partition, data partitions, . . .
  - Allows **grouping** of multiple slots (rootfs, appfs) as update targets
- **Bootloader interface supports common bootloaders**

  - grub
  - barebox
    - Well integrated with bootchooser framework
  - u-boot
  - EFI
- Storage support:
  - ext4 filesystem
  - vfat filesystem
  - UBI volumes
  - UBIFS
  - raw NAND (using nandwrite)
  - squashfs
- Independent from update sources
  - **USB Stick**
  - Software provisioning server (e.g. **Hawkbit**)
- Controllable via **D-Bus** interface
- Supports data migration
- Several layers of update customization
  - Update-specific extensions (hooks)
  - System-specific extensions (handlers)
  - Fully custom update script
- Build-system support

Yocto support in meta-rauc	PTXdist support since 2017.04.0.

Index