RAUC Documentation

Release v0.1.1

Jan Luebbe, Enrico Joerns, Juergen Borleis

Sep 21, 2017

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.

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.

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.

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.

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:

1. a secure channel to transfer the update or
2. 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.

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.

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.

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.

To pack this all together, the default bundle format uses SquashFS. 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.

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.

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.

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 accesed with, the type of storage or filesystem to use, its identification from the bootloader, etc.

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 accidently 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 togehter 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.

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 to a writable file system, RAUC will write an additional slot status file after having completed the write operation successfully. This file contains the slots hash.

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 file, if possible. If this hash equals the hash of the image to write, RAUC will skip updating this slot as a performance optimization.

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, RAUC will save update time by skipping the root file system automatically and only installing the changed application.

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.

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 prepartation 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.

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.

Creating Bundles

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

```
rauc bundle --cert=<certfile> --key=<keyfile> <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. `<output-file>` must be the path of the bundle file to create. Note that RAUC bundles must always have a `.raucb` file name suffix in order to ensure that RAUC treats them as bundles.

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.

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 via D-Bus.

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 [--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. Alternatively you can obtain a shell-parsable description, or a JSON representation of the system status.

Resigning Bundles

Note: This feature is not fully implemented yet

RAUC allows to resign a bundle from your build host, e.g. for making a testing bundle a release bundle that should have a key that is accepted by non-debugging platforms:

```
rauc resign --cert=<certfile> --key=<keyfile> <input-file> <output-file>
```

Reacting to a Successfully/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 explicitely 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*.

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 explicitely 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 explicitely 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, this can be remedied by choosing `booted` as the argument which is, by the way, the default if the optional argument has been omitted.

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, handlers and hooks.

The first is configuration through variables. This allow controlling the update in a predefined way.

The second type is using *handlers*. Handlers allow 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, handlers and hooks will be explained in more detail.

System Configuration File

Beside providing the basic slot layout, RAUC's system configuration file 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.

System-Based Customization: Handlers

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.

Install Handler

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

The install handler is the most powerful one RAUC provides. If you use this, you replace the entire default update procedure of RAUC. It will be executed between the pre-install and post-install handlers.

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.

Bundle-Based Customization: Hooks

Unlike handlers, 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.

The following environment variables will be passed to the hook executable:

RAUC_SYSTEM_COMPATIBLE The compatible value set in the system configuration file

RAUC_MF_COMPATIBLE The compatible value provided by the current bundle

RAUC_MF_VERSION The value of the version field as provided by the current bundle

RAUC_MOUNT_PREFIX The global RAUC mount prefix path

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 "Comptaible 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`.

The following environment variables will be passed to the hook executable:

RAUC_SLOT_NAME The name of the currently installed slot

RAUC_SLOT_CLASS The class of the currently installed slot

RAUC_SLOT_DEVICE The device of the currently installed slot

RAUC_SLOT_BOOTNAME If set, the bootname of the currently installed slot

RAUC_SLOT_PARENT If set, the parent of the currently installed slot

RAUC_SLOT_MOUNT_POINT If available, the mount point of the currently installed slot

RAUC_IMAGE_NAME If set, the file name of the image currently to be installed

RAUC_IMAGE_DIGEST If set, the digest of the image currently to be installed

RAUC_IMAGE_CLASS If set, the target class of the image currently to be installed

RAUC_MOUNT_PREFIX The global RAUC mount prefix path

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

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
```

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 system.conf *[handler]* section description on how to achieve this.

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 `/`.

The D-Bus API's main purpose is to trigger and monitor the installation process via its `Installer` interface. While the `Install` operation starts the installation progress, constant progress information will be emitted in form of changes to the `Progress` property. Upon completing the installation RAUC emits the `Completed` signal indicating either successful or failed installation.

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.

Examples Using `busctl` Command

Triggering an installation:

```
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer Install s "/path/to/
↪bundle"
```

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
```

Monitor the D-Bus interface

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

CHAPTER 4

Examples

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.

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.

RAUC Configuration

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

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

```
[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.

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
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.

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

[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.

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.

Example BSPs

- Yocto
- PTXdist

CHAPTER 5

Scenarios

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)

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`.

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
```

```
[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.

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*.

RAUC System Configuration

The system configuration file is the central configuration in RAUC that maps

RAUC expects the 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).

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

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
...
```

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.

In kernel Kconfig you have to enable the following options:

```
CONFIG_BLK_DEV_LOOP=y
CONFIG_SQUASHFS=y
```

Required Target Tools

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

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 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 is enabled:

- `CONFIG_FEATURE_TAR_AUTODETECT=y`

ext2/3/4 mkfs.ext2/3/4 (from e2fsprogs)

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 devices `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
```

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 informations about priority and remaining boot attemps 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 bootchooser.default="bootchooser"
```

Now, when Barebox is initialized it starts the bootchooser logik 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 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"
```

These virtual boot targets you connect to 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 possibilites in fine-tuning the behavior according to you 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 botchooser 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 device tree 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>;
                };
        };

        [...]
};
```

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 except 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 refer the see the official Barebox State Framework user documentation as well as the corresponding device tree 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 bootchoosers state variables from userspace, RAUC uses the tool *barebox-state* form the dt-utils repository.

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 RAUCs 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 boot slot (based on the slots *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
```

U-Boot

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

To enable handling of redundant booting in U-Boot, manual scripting is required.

The U-Boot bootloader interface of RAUC will rely on setting the U-Boot environment variables `BOOT_<bootname>_LEFT` which should mark the number of remaining boot attempts for the respective slot in your bootloader script.

To enable reading and writing of the U-Boot environment, you need to have the U-Boot target tool `fw_setenv` available on your devices rootfs.

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

Note: If you want to implement different behavior or use other variable names, you will need to modify the `uboot_set_state()` and `uboot_set_primary()` functions in `src/bootchooser.c`.

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`.

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

An examplary 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.

Others

It is planned to add support for a *custom* boot selection implementation that will allow you to use also non-conventional or yet unimplemented approaches for selecting your boot slot.

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
```

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 systems service file folders.

It is a good idea to wait for the system to be fully started before marking it as succesfully booted. In order to achieve this, a smart soluting is 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
```

D-Bus Integration

The D-Bus interface RAUC provides makes it easy to integrate it into you custom application. 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, make sure the D-Bus service file `de.pengutronix.rauc.service` from the `data/` dir gets installed properly.

Watchdog Configuration

Detecting system hangs during runtime requires to have a watchdog and to have the wathdog 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
```

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 layers README file.

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:"

SRC_URI_append := "file://system.conf"
```

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.

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*.

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
```

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-pbsp/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_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: `rootfs` (default), `kernel`, `bootloader`.

Based on this information, your bundle recipe will build all required components and generate a bundle from this. The created bundle can be found in `tmp/deploy/images/<machine>/bundles` in your build directory.

PTXdist

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

Integration into Your RootFS Build

To enable building RAUC for your target, set:

```
CONFIG_RAUC=y
```

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

You should also customize the compatible RAUC uses for your System. For this set `CONFIG_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 `configs/platform-<yourplatform>/projectroot/etc/rauc/system.conf` to let the RAUC recipe install it into the rootfs you build. Also place the keyring for your device in `configs/platform-<yourplatform>/projectroot/etc/rauc/ca.cert.pem`.

Note: You should use your local PKI infrastructure for generating valid certificates and keys for your target. For debugging and testing purpose, PTXdist provides a script that generates a set of example certificates. It is named `rauc-gen-test-certs.sh` and located in PTXdist's `scripts` folder.

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 systemd and reset by *rauc status mark-good* to indicate a successful system startup.

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 systems rootfs. As for all image recipes, the *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.

In order to sign your update (mandatory) you also need to place a valid certificate and key file in your BSP at the following paths:

```
$(PTXDIST_PLATFORMCONFIGDIR)/config/rauc/rauc.key.pem (key)
$(PTXDIST_PLATFORMCONFIGDIR)/config/rauc/rauc.cert.pem (cert)
```

Once you are done with you setup, PTXdist will automatically create a RAUC update Bundle for you during the run of `ptxdist images`. It will be placed under `<platform-builddir>/images/update.raucb`.

CHAPTER 7

Advanced Topics

Security

The RAUC bundle format consists of a squashfs image containing the images and the manifest, which is followed by a public key signature over the full image. This signature is stored in the CMS (Cryptographic Message Syntax, see RFC5652) format. Before installation, the signature is verified against the keyring already stored on the system.

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 and 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. An PKI with intermediate CAs for the unit tests is generated by the `test/openssh-ca.sh` shell script, 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.

CA Configuration

OpenSSL uses a `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 (espcially ca, req, x509, cms, verify and config) and Stefan H. Holek's pki-tutorial are useful.

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

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.

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.

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.

Storing Data in The Root File System

In case of a writeable 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.

Using Data Partions

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 you 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.

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 format 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!)

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. For example eMMCs have two dedicated boot partitions (see the JEDEC standard JESD84-B51 for details), one of which can be enabled atomically via configuration registers in the eMMC.

As a further example, 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.

Currently, independent of whether you are able to update your bootloader with fallback, atomically or with some risk of an unbootable system, our suggestion is to handle updates for it outside of RAUC. The main reason is to avoid

booting an old system with a new bootloader, as this combination is usually not tested during development, increasing the risk of problems appearing only in the field.

One possible approach to this is:

- 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.

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.

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 sytem. 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 interpretate and appropriately handle a bundle with the latest options

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?

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.

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.

The RAUC hawkBit client

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 and testing it, 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.

General

- System compatible is specific enough
- Bundle version policy defined
- Bundle contains all software components
- Bundles are created automatically by a build system
- Bundle deployment mechanism defined (pull or push via the network, from USB/SD, . . .)

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)

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)

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

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

Security

- PKI configured
- Certificate validity periods defined
  - Systems always have correct time *or*
  - Validity period is large enough
- Key revocation tested
- Key rollover tested
- Separate development and release keys deployed
- Per-user or per-role keys deployed

Data Migration

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

CHAPTER 9

Frequently Asked Questions

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.

CHAPTER 10

Reference

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.

Example configuration:

```
[system]
compatible=FooCorp Super BarBazzer
bootloader=barebox

[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`, `u-boot`.

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

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

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*.

[keyring] section

The `keyring` section refers to the trusted keyring used for signature verification.

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

[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.

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.

device The slot's device path.

type The type describing the slot. Currently supported values are `raw`, `nand`, `ubivol`, `ubifs`, `ext4`, `vfat`. See table *Slot Type* for a more detailed list of these different types.

bootname For bootable slots, the name the bootloader uses to identify it. The real meaning of this depends on the bootloader implementation used.

parent The `parent` entry is used to bind additional slots to a bootable root file system slot. 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>`.

readonly 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.

Manifest

A valid manifest file must have the file extension `.raucm`.

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

[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.

[hooks] section

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

hooks List of hooks enabled for this bundle.

[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).

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.

Slot Status File

A slot status file is generated by RAUC after having updated a slot. If the slot is writeable for RAUC (because it contains a writable filesystem), it will place a small file named `slot.raucs` in its root directory, containing the sha256 of the installed image.

```
[slot]
status=ok
sha256=b14c1457dc10469418b4154fef29a90e1ffb4dddd308bf0f2456d436963ef5b3
```

Command Line Tool

```
Usage:
  rauc [OPTION...] <COMMAND>

Options:
  -c, --conf=FILENAME               config file
  --cert=PEMFILE                    cert file
  --key=PEMFILE                     key file
  --keyring=PEMFILE                 keyring file
  --mount=PATH                      mount prefix
  --override-boot-slot=SLOTNAME     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
  checksum      Update a manifest with checksums (and optionally sign it)
  install       Install a bundle
```

```
  info          Show file information
  status        Show status
```

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_UPDATE_SOURCE Path to mounted update bundle, e.g. `/mnt/rauc/bundle`

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_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
```

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

Methods

Install (IN s source);

Signals

Completed (i result);

Properties

Operation readable s

LastError readable s

Progress readable (isi)

Description

Method Details

The Install() Method

```
de.pengutronix.rauc.Installer.Install()
Install (IN  s source);
```

Triggers the installation of a bundle.

IN s *source*: Path to bundle to be installed

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)

Property Details

The "Operation" Property

```
de.pengutronix.rauc.Installer:Operation
Operation  readable   s
```

Represents the current (global) operation RAUC performs.

The "LastError" Property

```
de.pengutronix.rauc.Installer:LastError
LastError  readable   s
```

Holds the last message of the last error that occured.

The "Progress" Property

```
de.pengutronix.rauc.Installer:Progress
Progress  readable   (isi)
```

Provides installation progress informations in the form

(percentage, message, nesting depth)

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
      i. Skip update if new image hash matches hash of insalled 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

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 (fatory 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.

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.

Code

- Basically follow the Linux kernel coding style

Documentation

- Use semantic linefeeds in .rst files.

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:
>
> 1. 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
> 2. 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
> 3. The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
> 4. 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

In Development

Documentation

- Added docs/, CHANGES and README to tarball

Release 0.1.1 (released May 11, 2017)

Enhancements

- systemd service: allow systemd to manage and cleanup RAUCs mount directory

Bugs fixed

- 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

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

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.

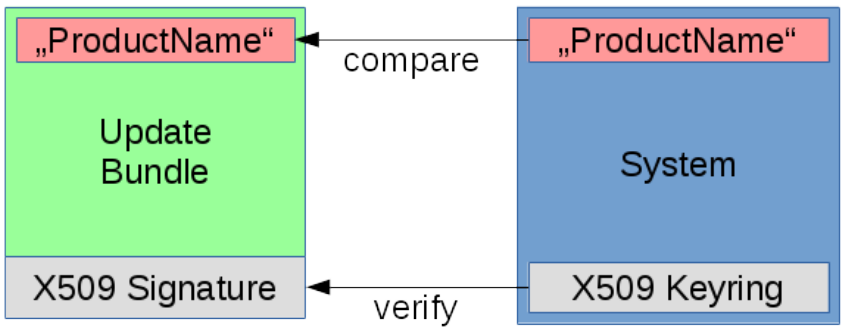
RAUC does NOT intend to be a deployment server. On your host side, it only creates the update artifacts. You may want to have a look at rauc-hawkbit for interfacing with the hawkBit deployment server.

And finally, factory bring up of your device, i.e. initial partitioning etc. is also out of scope for and 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)



- **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
- Storage support:
  - ext2/3/4 filesystem
  - UBI volumes
  - UBIFS
  - raw NAND (using nandwrite)
  - squashfs
- Independent from update source
  - **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