RAUC Documentation

Release v0.1.1

Jan Luebbe, Enrico Joerns, Juergen Borleis

Jun 14, 2017

Contents

Contents:

CHAPTER 1

Overview

The Need for Updating

Updating an embedded system is always a crucial 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 plugging in a special USB stick or by 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.

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 several 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-proven, solid and generic base for the different custom requirements and restrictions an update concept for a specfic 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.

Key Features of RAUC

- **Fail-Safe & Atomic**:
  - An update may be interrupted at any point without breaking the running system.
  - Update compatibility check
- **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
- 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 updates source
  - **USB Stick**
  - Software provisioning server (e.g. **Hawkbit**)
- Controllable via **D-Bus** interface
- Supports data migration
- Network protocol support using libcurl (https, http, ftp, ssh, ...)
- Several layers of update customization
  - Update-specific extensions (hooks)
  - System-specific extensions (handlers)
  - Fully custom update script
- Yocto support in meta-rauc

Redundant Updates

Being able to safely update an entire system with pre-defined images normally requires more than one bootable device or partition available. A minimal setup would consist of a running system on slot A and an inactive system on slot B. A bootloader is responsible for booting the desired system.

Now, the running system may perform an update on the inactive slot B. Once the update was performed successfully, the system must tell the bootloader to boot from slot B from now on. To add more safety, a third bootable slot C may be used containing a minimal fall-back system the bootloader may choose if booting from the other slots fails. This one might also be used to initially install a production system on a new device.

In the following an overview of the basic concept RAUC uses for realizing such an update system is provided.

Slots

RAUC's view of the target system it is running on is described using so-called slots. Slots are possible targets for (parts of) updates. Usually, they are partitions on an SD/eMMC, UBI volumes on NAND/NOR flash or raw block devices. The system designer must provide a configuration file that lists all slots that RAUC should use and describe which device they are stored on, how the bootloader may detect them, etc.

Bundles

An update bundle is a squashfs-packed set of config files, scripts, and disk images with an appended signature that allows verifying the bundle's origin and integrity.

Booting

To determine from which slot the system is booted, the bootloader must provide a *boot chooser*. This allows maintaining multiple boot sources with a *defined priority*, a *number of boot attempts*, and a flag to deactivate the source.

If booting from the highest-priority system (typically the current production system) fails for e.g. 3 times, the next lower priority boot source is chosen (which could be the fallback system).

As updates are always installed in a currently inactive slot, the boot priority must be changed after a successful update.

Basic Update Procedure

The RAUC service that runs on the target will perform an update when being triggered by an install command providing an update bundle. An update request may be initiated manually from the command line, via D-Bus or by a script that checks for example for insertion of an USB stick containing a firmware bundle. Then the default (and simplified) update behavior will be the following:

1. RAUC verifies the bundle by checking its signature against the keyring located in the root file system. A bundle with an invalid signature will be rejected.
2. RAUC mounts the bundle (which simply is a squashfs image)
3. Verify bundle compatibility:
    - The compatible string in the manifest is compared against the compatible string stored in the system configuration file.

- If the strings are different, the bundle will be rejected to prevent installing an incompatible bundle.

4. Determine the target *install group*, i.e. which slots an update will be installed to.
7. Mark target slots as non-bootable for bootloader.
6. Iterate over each image specified in the manifest
   - Try to read slot status informations.
   - If the provided slot image is different from the installed one: Update slot with a method determined by the type of slot and the image type.
   - Try to write slot status informations.
7. Mark target slots as new primary boot source for the bootloader.
8. Terminate successfully if no error occurred.

Once the update controller receives an update request instruction containing the file path of a firmware bundle it verifies its signature based on a public key stored in the current rootfs. If the signature is valid, the service loopback-mounts the bundle to access its content and installs the update.

Installing the update means either calling an *update handler* included in the bundle (if provided) or using a default handler that performs the update based on information about the available slots and versions.

Target Slot Selection

The *boot chooser* (in the bootloader) passes the name of the booted slot using the kernel command line. This allows the *controller* to identify the currently active slots.

To select the target slot, the controller first looks for a slot marked as non-bootable. This could be caused by an interrupted update or repeated boot failures.

If no non-bootable slot exists, the inactive slot with the lowest priority is selected.

CHAPTER 2

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.

Some hints are already useful when designing the device itself, here you can set the base for a great embedded system. For some other systems it might not be (completely) in your hand to select the best hardware. But don't fear, we can also deal with these cases.

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.

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

Update Source and Provisioning

USB Stick to deployment server.

Security

An update tool 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.

CHAPTER 3

Using RAUC

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

- One for your host (development) system, where you create new updates.
- One that will run on the target for performing updates.

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 in a way the RAUC updater on the target can handle it and knows 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 make RAUC treat them as bundles.

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 productive bundle that should have a key that is accepted by non-debugging platforms:

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

Obtaining Bundle Information

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

You can control the output type of RAUC info depending on your needs. By default it will print a human readable representation of the bundle. Alternatively you can obtain a shell-parsable description, or a JSON representation of the bundle content.

Installing Bundles

To 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.

See System Status

For debugging purposes and for scripts maybe it can be helpful to gain an overview over 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.

Marking Boot as Successful / Failed

Normally, the full system update chain is not completed before being sure the newly installed system runs without any errors. As the definition and detection of a *successful* operation is really system-depended, RAUC provides a command for marking a boot as either successful or failed.

```
rauc status mark-good
```

This marks a boot as successful for the underlying bootloader implementation. This will, for example, reset a boot attempt counter.

```
rauc status mark-bad
```

This marks a boot as failed for the underlying bootloader implementation. In most cases this will disable the currently booted slot or at least switch to another one.

Customizing the Update

RAUC provides several ways to customize the update process. Some allow to add and extend details more fine-grainedly, some allow to replace 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 pre-defined variables. This allows to control the update in a predefined way.

The second type is using *handlers*. Handlers allow to extend or replace 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 all future updates.

The last type are *hooks*. They are much like *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 system 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-Based Customization: Handlers

- system.conf
- multiple scripts?

For a detailed list of all environment variables exported for the handler scripts, see ...

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 and its content accessible, 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 has. If you provide this, you replace the entire default update procedure of RAUC. It will be executed right after the pre-install handler and right before the post-install handler.

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 hooks 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 currently installed 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
```

Using the D-Bus API

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 *lasterror* property, which contains the last error that occured during an installation.

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

Monitor the D-Bus interface

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

Migrating to an Updated Bundle Version

RAUC will be constantly extended and new features or enhancements will make their way into RAUC. Thus, also the information contained in the bundle, or, more precise, in the manifest may extend.

Now, current RAUC versions check each option contained in the manifest and will reject the bundle if an entry in the manifest is not known. This is necessary and important to assure that the actual installation behavior does not differ from the description in the manifest.

Despite the fact, that installing bundles that use newly added features will *not work*, it is (of course) still possible to update your device to this new version.

For this you have to follow a simple **2 step process**

1. Create a bundle not containing the new RAUC features in the manifest but include the new RAUC version itself in the rootfs image.
2. Now you can proceed updating your board with bundles that use the latest RAUC features.

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.

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

Asymetric Slots

- One full + one rescue
- Useful if not enough space for two full installations

Multiple Slots

- rootfs + appsfs
- Useful if the application should be updated independently of the base system

Additional Rescue Slot

- Two full + one rescue
- Rescue slot is not touched by normal updates
- Rescue slot is booted if both normal slots fail to boot

CHAPTER 6

Design Checklist

Watchdog vs. Confirmation

- WD must reset the whole system (DVFS, flash state)

Symmetric vs. Rescue+Normal

Image Signing

- Development + Release
- PKI
- Certificate Revocation

Configuration

Most systems require a location for storing configuration data. Unlike for example the root or application filesystems which are often mounted read-only, a configuration partition would be writable to allow modifying configuration data.

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

- May configuration format change over different application versions?
-

CHAPTER 7

Integration

When integrating RAUC (and in general) we recommend using a Linux system build tool like Yocto / OpenEmbedded or PTXdist. For information about how to integrate RAUC using these tools, refer to section *Yocto* or *PTXdist*.

System Configuration

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 todo. 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 group used for redundancy and *slot-index* is the index of the individual slot starting with 0. If you have two rootfs slots, for example, one slot section will be named `[slot.rootfs.0]`, the other will be named `[slot.rootfs.1]`. RAUC does not have predefined class names. The only requirement is that the class names used in the system config match those 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, and the `bootname` which is the name the bootloader uses to refer to this slot device.

Type

A list of common types supported by RAUC:

Type	Description
raw	A partition holding no (known) file system. Only raw image copies may be performed.
ext4	A partition holding an ext4 filesystem.
nand	A NAND partition.
ubivol	A NAND partition holding an UBI volume
ubifs	A NAND partition holding an UBI volume containing an UBIFS.

Kernel Configuration

The kernel used on the target device must support both loop devices and the SquashFS file system to allow installing 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 allow RAUC handling a bootable slot, you have to mark it bootable in your system.conf and configure the name under which the bootloader is able to identify this distinct slot. This is both done by setting the `bootname` property.

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

Barebox

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

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
```

To enable reading and writing of the required state variables, you also have to add the `barebox-state` tool from the dt-utils repository to your systems rootfs.

Note: For details on how to set it up, which storage backend to use, etc. refer to the Barebox bootchooser documentation.

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

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

System Boot

- Watchdog vs. Confirmation
- Kernel Command Line: booted slot
- D-Bus-Service vs. Single Binary
- Cron

Backend

Persistent Data

- SSH-Keys?

Feel free to extend RAUC with support for your bootloader.

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 **'chapter'_**.

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 8

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`.

[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.

[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 script will be called ...

pre-install This script will be called ...

post-install This script will be called ...

[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`. See table todo for a more detailed list about 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>

Application Options:
  -c, --conf=FILENAME     config file
  --cert=PEMFILE          cert file
  --key=PEMFILE           key file
  --mount=PATH            mount prefix
  --handler-args=ARGS     extra handler arguments
```

```
  --version               display version
  -h, --help

List of rauc commands:
  bundle        Create a 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
```

Signatures

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)

CHAPTER 9

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 10

Design Decisions

Bundle Mode

- squashfs
  - mountable
  - avoids copies
  - signature location
  - easy to use for USB memory sticks or upload via a web interface

Network Mode

- manifest should define complete consistent system
- manifest is signed with a detached CMS signature
- manifest contains size and cryptographic hash of each file
- RAUC detects files which have not changed and skips the download
- server can control update process

Custom Handlers

- a handler can override default RAUC behaviour to handle special cases (which were not anticipated during the initial project development)
  - sanity checks
  - modifications to the partition layout

- bootloader updates

CHAPTER 11

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 12

Changes in RAUC

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

Index