Case Study:

Switching from asymmetric to symmetric software updates

Jeff Pautler

jeffrey.pautler@ni.com,

About me

- I work for NI (formerly known as National Instruments)
  - Makes hardware & software for test, measurement, and automation

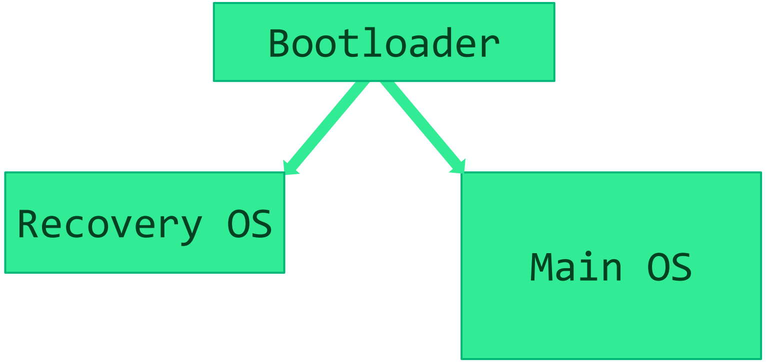
- Real-Time OS group for the past 5 years
  - Maintain the real-time Linux distribution for our hardware
  - PREEMPT_RT based Linux kernels
  - Embedded 32-bit ARMs and x86_64 systems
  - Distribution based on OpenEmbedded/Yocto

Agenda

- Asymmetric vs Symmetric Updates
- Use Case Background
- Motivations
- Requirements
- Implementation

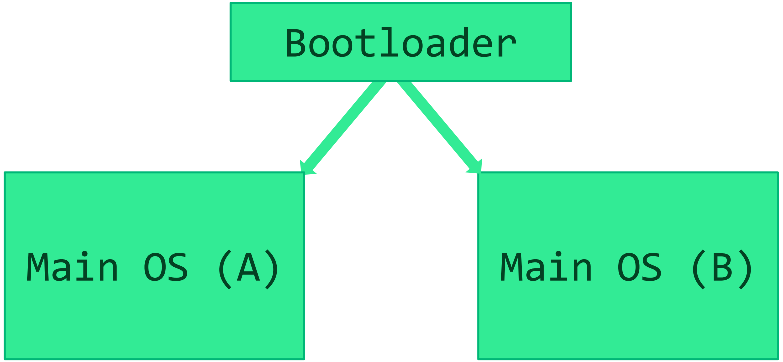
Asymmetric Updates

- Two different environments
- Main OS (normal operating environment)
- Recovery OS (recovery environment)
  - Used to update the Main OS
  - Rarely needs updates
  - Often uses an initramfs to run a non-persistent root filesystem from memory



Symmetric Updates

- Two copies of the Main OS
- The booted copy is "active"
- Update is written to the "inactive" copy
- If updated copy fails to boot, then fall back to unmodified copy
- After successful update, retain unmodified copy or replace with updated copy for later fallback



Symmetric Updates & Bootloader

- After successfully writing an updated image, tell the bootloader to load that image just once on the next boot
  - Need to instruct the bootloader from userspace
- Bootloader attempts to boot the new image
- If booting the new image is successful, the system informs the bootloader to put the updated image first in the boot order
- If booting the new image fails, the bootloader will boot the old image on the next reboot since the boot order was not changed

Use Case Background

- Our Recovery OS has more than basic upgrade functionality
  - Configure network settings, password, etc
  - Respond to network discovery requests
  - Erase Main OS and data partition (“format”)
  - Serves as fallback when something goes wrong in Main OS
- Recovery OS was intended to update infrequently
  - Reality has been that it changes every 1-2 years
- Failure during update of Recovery OS can leave system unbootable
  - Requires hands-on reprovisioning via USB

Use Case Background

- Our Main OS image is not fixed
  - Users can make changes
    - Install additional software via package manager
    - Install additional expansion hardware and corresponding drivers
    - Install their own application running at Real-Time priority
    - Change system configuration
    - Basically anything, as they have root access to the system

- It is possible for users to put the Main OS in an unbootable state

Motivations

- Eliminate cost of separate Recovery OS and Main OS images
  - Separate build configurations/source
  - Debugging build failures
  - Validating images
- Replace in-house update logic with open source tooling
- Make updates fail-safe

Requirements

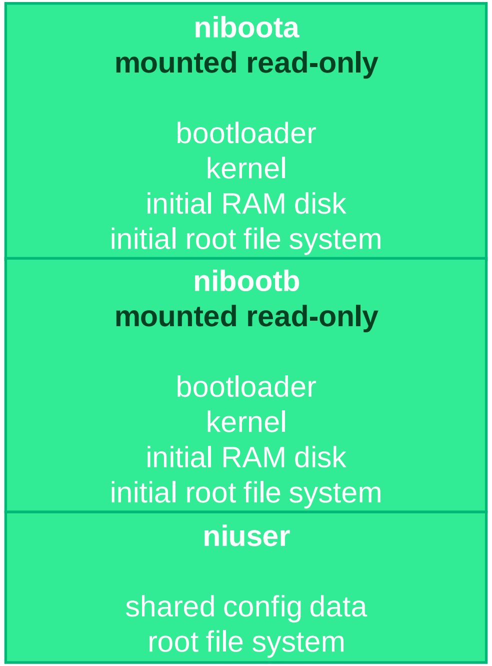
- Updates are fail-safe
- Preserve some configuration across updates
  - Device identity, network settings, password database, etc
- Within reason, system should remain bootable despite changes made by user
- OS can be reset to original state
  - Undo installed packages
  - Undo configuration changes
  - Reset to state when last updated

Restructure Partition Contents

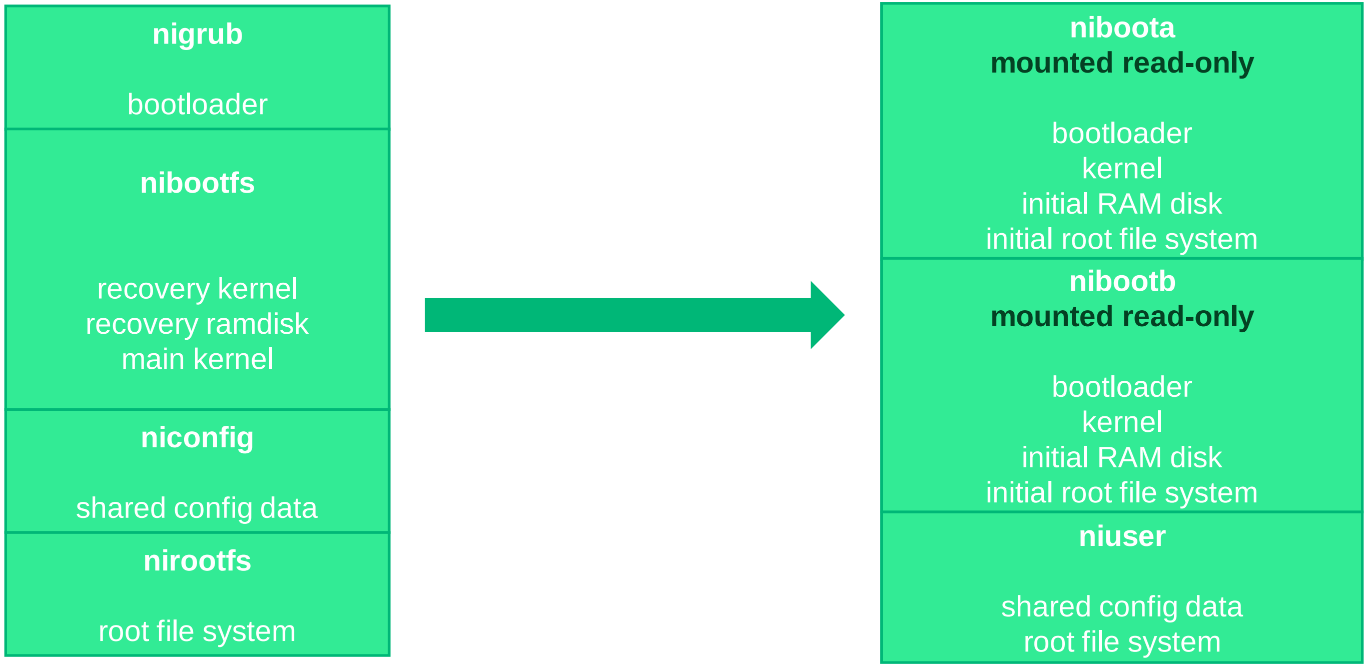
- Move everything required to boot into same partition
  - Bootloader
  - Kernel
  - Initial RAM disk
  - Initial root file system
    - Compressed read-only file system (squashfs)

- Allows updating an image by replacing contents of one partition

- Contents of active boot partition will not change while running, so it can be mounted read-only



Restructure Partition Contents



Robust Auto Update Controller (RAUC)

- RAUC is the tool we use to manage boot partitions and their contents
  - Builds update artifacts into "bundles"
  - Installs a bundle to a "slot"
  - Interfaces with EFI to:
    - Mark slot as "boot next“
    - Modify boot order

- RAUC is an open source project - https://rauc.io/

Robust Auto Update Controller (RAUC)

- Provide system information to RAUC in a configuration file
  - Specify bootloader interface
    - Our systems use EFI bootloader
  - Slot definitions

```
# RAUC configuration file

[system]
compatible=nilrt-efi-ab
bootloader=efi

# system partition A
[slot.niboot.0]
device=/dev/sda1
type=vfat
bootname=niboota

# system partition B
[slot.niboot.1]
device=/dev/sda2
type=vfat
bootname=nibootb
```

Robust Auto Update Controller (RAUC)

- Create RAUC bundles using Yocto bundle bbclass provided by meta-rauc

```
DESCRIPTION = "NILRT system bundle containing runmode image"
…
BUNDLE_IMAGE = "nilrt-dkms-bundle-image"
DEPENDS = "${BUNDLE_IMAGE}"
LICENSE_CREATE_PACKAGE = "0"

SRC_URI += "file://nilrt-bundle-hooks.sh"

RAUC_BUNDLE_COMPATIBLE = "nilrt-efi-ab"
RAUC_BUNDLE_DESCRIPTION = "${DESCRIPTION}"
RAUC_BUNDLE_VERSION = "${BUILDNAME}"
RAUC_BUNDLE_BUILD = "${BUILDNAME}"

RAUC_BUNDLE_HOOKS = "1"
RAUC_BUNDLE_HOOKS[file] = "nilrt-bundle-hooks.sh"
RAUC_BUNDLE_HOOKS[hooks] = "install-check;"

RAUC_BUNDLE_SLOTS = "niboot"
RAUC_SLOT_niboot = "${BUNDLE_IMAGE}"
RAUC_SLOT_niboot[fstype] = "tar.bz2"
RAUC_SLOT_niboot[hooks] = "pre-install;post-install;"

RAUC_SIGN_BUNDLE = "0"
inherit bundle
```

Robust Auto Update Controller (RAUC)

- Install a bundle by calling RAUC directly:
  - **rauc install “bundle.raucb”**
    - This call installs the bundle and marks the new slot as “boot next” in EFI
    - “Boot next” causes a one-time boot to that slot on the next reboot

- After successfully booting an updated image, an init script calls RAUC to mark the contents of the currently booted slot as good:
  - **rauc status mark-good**
    - This call changes the boot order in EFI

- If updated image fails to boot, the EFI boot order is left unchanged
  - Next boot will fall back to the unchanged boot slot

Provisioning

- Blank drives are provisioned from a USB thumb drive running a Linux image
  - Image contains the necessary tools and the initial Main OS image
- During init, this image runs a provisioning script
- The provisioning script creates the niboota, nibootb, and niuser partitions
  - Boot partitions are EFI partitions
- EFI boot entries are created for niboota and nibootb

```
efibootmgr -c -d "$TARGET_DISK" -p 1 -L 'niboota' -l '\efi\nilrt\bootx64.efi’
efibootmgr -c -d "$TARGET_DISK" -p 2 -L 'nibootb' -l '\efi\nilrt\bootx64.efi'
```

Provisioning

- Run RAUC to install initial Main OS bundle to both boot partitions

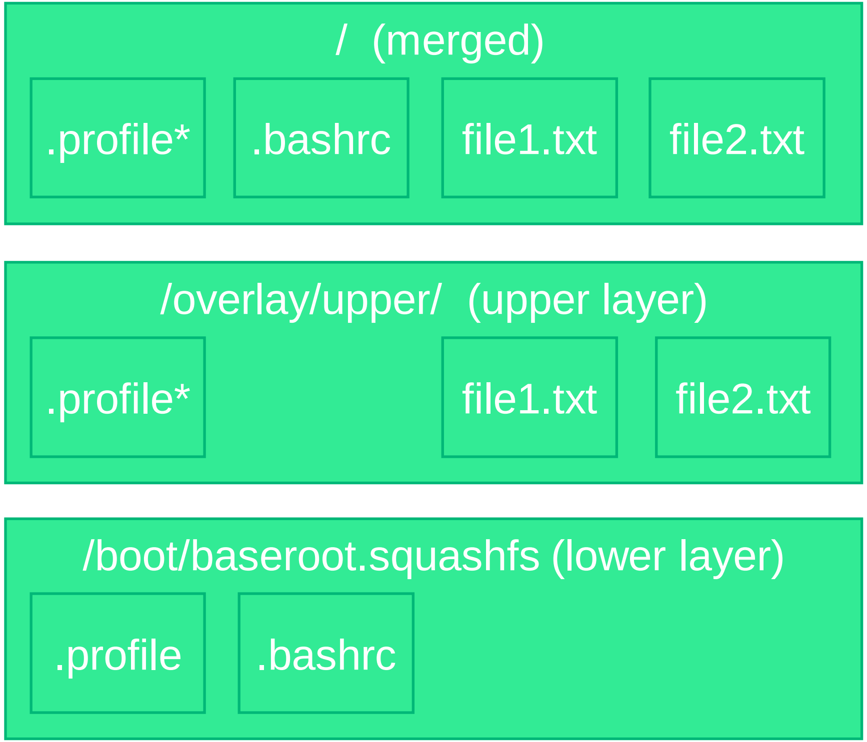
```
# Override current boot slot to be niboota, so RAUC installs to nibootb slot
rauc --override-boot-slot=niboota install /payload/niboot.raucb

# Mark-good nibootb to place it at front of EFI BootOrder
rauc --override-boot-slot=nibootb status mark-good niboot.1

# Override current boot slot to be nibootb, so RAUC installs to niboota slot
rauc --override-boot-slot=nibootb install /payload/niboot.raucb

# Mark-good niboota to place it at front of EFI BootOrder
rauc --override-boot-slot=niboota status mark-good niboot.0

# Finally, mark-active niboota to ensure it's EFI BootNext
rauc --override-boot-slot=niboota status mark-active niboot.0
```

Overlay Filesystem

- Initial root file system is read-only
- Need to write new files
- Need to change files in initial root filesystem
- Use Linux overlay file system (OverlayFS)

Overlay Filesystem

- OverlayFS combines a lower layer with an upper layer and presents a merged result
- In our case, the lower layer is the read-only initial root file system from the boot partition
- Our upper layer is a directory tree of files in the niuser partition
- OverlayFS is transparent to applications



Overlay Filesystem

- Upper layer holds newer versions of files in the lower layer. If files in the lower layer change, there can now be conflicts with the copies in the upper layer.

- Need to reset the overlay (discard the upper layer) when lower layer changes
  - Record checksum of underlying initial root file system when overlay is created
  - Recalculate checksum of initial root file system file on each boot
  - If checksums do not match, then reset the overlay
    - This means creating an empty upper layer and recording a new checksum of the underlying initial root file system
    - Old overlay is not actually discarded until after next successful boot in case we need to fall back

Overlay Filesystem

- Overlay handling is done mostly in init script in initial RAM disk
  - Checksums are calculated and compared
  - Overlay is created/reset
  - Overlay is mounted
  - switch_root to the mounted overlay

- Overlay is reset when booting an updated image
  - Because checksums no longer match

- Overlay is not reset when installing and booting same image
  - This includes falling back to the previous image during a failed update

- Overlay reset is a fast and easy way to revert image to state before any changes were made
  - Previously, this required reformatting and reinstalling the image

Overlay Filesystem

- Code to setup the overlay

```
mkdir -p "$U_MNT/overlay/lower"
mkdir -p "$U_MNT/overlay/upper"
mkdir -p "$U_MNT/overlay/work"
mkdir -p "$U_MNT/overlay/image"

# Mount lower filesystem
mount -o ro -t squashfs "$B_MNT/baserootfs.squashfs" "$U_MNT/overlay/lower"

# Create overlay image
mount -t overlay -o lowerdir=$U_MNT/overlay/lower,\
                    upperdir=$U_MNT/overlay/upper,\
                    workdir=$U_MNT/overlay/work \
      overlay "$U_MNT/overlay/image"

# Remove sync option from niuser mount in preparation for toggle
sync
mount -o remount,async "$U_MNT"

exec switch_root "$U_MNT/overlay/image/" /sbin/init $init_options
```

Transfer Configuration Scripts

- Updating an image must preserve the target identity and important settings
  - Hostname and network settings
  - Login credentials
  - SSH keys

- A main script saves and restores configuration data
  - Calls sub-scripts, one for each setting or group of settings

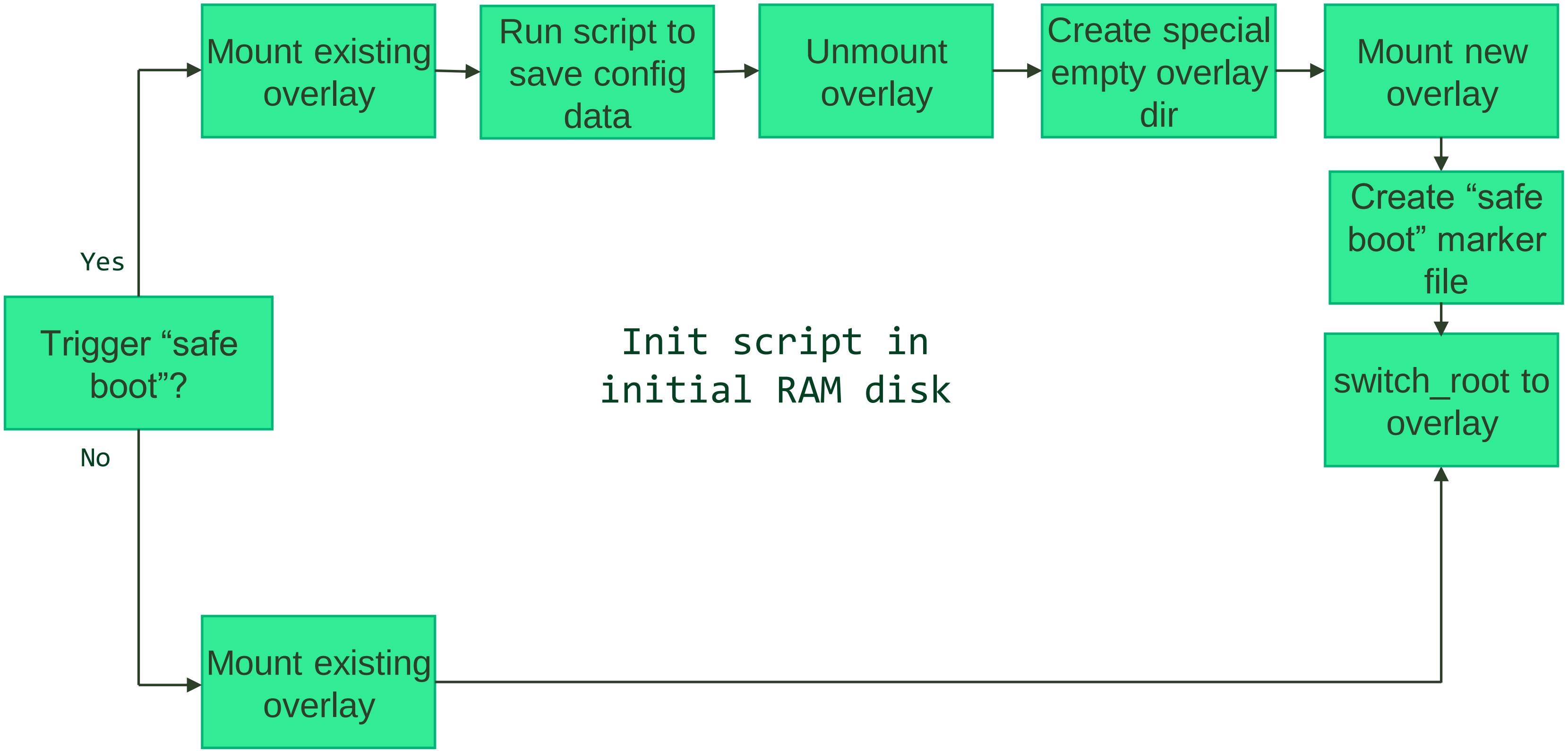
- Before updating to a new image or resetting the overlay the main script is run to save configuration data

- On first boot of a new overlay, the main script is run to restore data

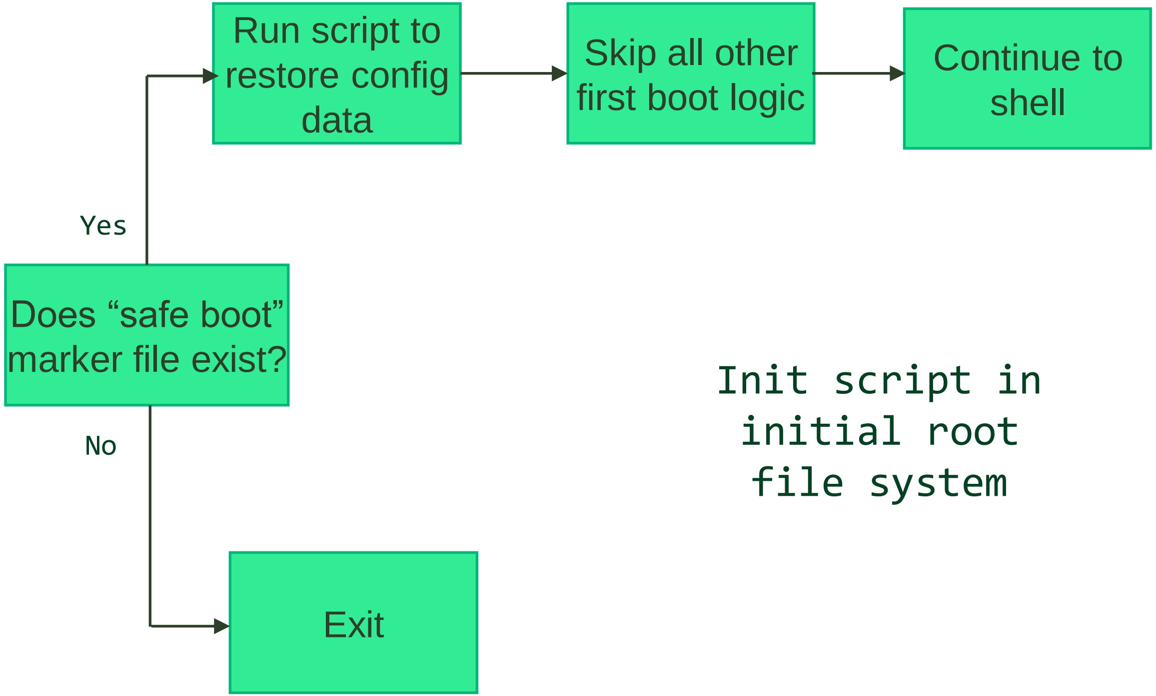
Need A Safe Boot State

- Because users can change the system, we want a way to boot to a known state
- Use this state for applying software updates
- Use this state as a fallback in case system will not boot normally
  - Could be detected by bootloader or watchdog timer
- Can also be triggered by user
  - Could be initiated via software
  - Button press during power-on can be detected during boot
- Leverage OverlayFS and transfer configuration scripts to accomplish this

Need A Safe Boot State



Need A Safe Boot State



Init script in initial root file system

Summary

- Asymmetric vs Symmetric Updates
- Robust Auto Update Controller (RAUC)
- Overlay Filesystem
- Using temporary overlays for safe booting