# There Is No Store For Self-Driving Car Parts

## Running the Ultimate Battery-Powered Device with Linux

Stephen Segal and Matt Fornero. Cruise LLC

Embedded Linux Conference North America 2020

# Agenda

1. About Us
2. The Vehicle
3. Challenges
4. Buildroot
5. Booting
6. Device Management

cruise

# About Us

# Cruise

- Majority-owned subsidiary of General Motors
- Additional investment from Honda and others
- Based and testing in San Francisco

We’re building the world’s most advanced self-driving vehicles to safely connect people with the places, things, and experiences they care about.

# Cruise Origin

A new all-electric vehicle designed from the ground up for autonomous ride-sharing.

- Designed to last for more than 1 million miles.
- Modular, so sensors and components can be easily replaced over time.
- Will be built for roughly half the cost of a conventional electric SUV.

# Cruise Embedded Systems

We provide the interface from the driving stack to the rest of the vehicle. This includes:

- Bringing up custom hardware
- Providing Linux OS images
- Developing application-level software for edge devices

Or for the less technically-inclined:

- We figure out how to get a supercomputer into a car.

cruise

# The Vehicle

# The Cruise Autonomous Vehicle (AV)

# The Cruise Autonomous Vehicle (AV)


We make a lot of custom hardware…

# A Heterogeneous High-Performance Computer on Wheels

- Sensors
  - Cameras
  - LiDAR
  - Lots more…
- In-car Networking Infrastructure
- Telematics
- Core Compute

# Suppliers

Our hardware comes from various sources

- Traditional Tier 1 automotive suppliers
- Internal designs
- Non-automotive IT vendors
- Components not related to autonomy are typically sourced by GM

Software is largely developed in-house

- This allows us to iterate quickly and ensure consistency.

# Challenges

# Cars and Linux

- Tier 1 automotive suppliers have little experience with Linux
  - Normally work with AUTOSAR and other RTOSes
  - Not used to developing hardware for a customer-provided Linux operating system.
- Automotive-qualified parts don't always provide Linux drivers or support
  - Automotive SoCs with Linux support are often limited to parts designed for infotainment systems.
  - Vendors of automotive-specific peripherals often don't provide Linux support at all.

# A Brave New World

Autonomous vehicles represent a globally-unsolved problem:

- Many of the components we're building have never been built or contemplated before
  - Often the components we really need (SoCs, peripherals, etc.) don't actually exist.
- Still, some are more typical. When possible, we want to leverage existing designs
  - We don't want to unnecessarily invent new stuff when we have so much necessary new stuff to invent.

# Hardware Capabilities

- Our devices range from large high-performance computers to small thermally and compute constrained components.
- Majority of autonomy-related components are x86-64 or ARMv8-A systems.
  - Some ARMv8-A SoCs contain an additional microcontroller core for functional safety and real-time operations
  - Traditional electronic control units (ECUs) are developed by our OEM partners.

# The Challenge

Ultimately we want to

1. Support a large number of diverse devices.
2. Ease development of new devices when the nature of such devices isn't always predictable.
3. Guarantee high and predictable reliability
4. Secure the on-vehicle computing infrastructure.

# The Tools

How do we do all this?

1. Enforce consistency when possible.
2. Use common repositories and tools.
3. Don’t invent things unnecessarily.
4. Design for maximum flexibility.

cruise

# Buildroot

# Why Buildroot?

We use Buildroot as a build system

- Goal: Build a firmware image, not a distro
- Simplicity: Easy onboarding for new developers
- Speed: Performant clean builds
- Sustainability: Simple support for internal package mirrors / monolithic repo
- Extensibility: Can be easily extended with new functionality

# Why Extend Buildroot?

- Must support building lots of different boards with varying requirements
  - Config layering to ensure a consistent set of applications
  - Allow developers to easily switch between boards.
    - Allow target selection, separating output directories per target
    - Ensure similar organization
  - Support automatic test execution as part of continuous integration.

# Repository Organization

- A top-level Makefile manages the system and allows selecting a current target
  - Manages output directories and concatenating the configuration layers
  - Provides targets for automatic download of images to a board
- Buildroot maintained as a git submodule.
  - We typically maintain a branch off each LTS release.
  - Move once a year.
- A local apps directory for storing simple applications and libraries that don't merit a separate repository.
- A place for out-of-tree kernel modules
- An output directory that contains build products: this is separated by build

```
.
├── apps
├── br2-external
├── buildroot
├── buildroot-cache
├── ci
├── docs
├── linux-kernel
├── Makefile
├── makefiles
├── output
├── OWNERS
├── ReadMe.md
├── remote
├── scripts
└── VERSION.txt
```

# br2-external

- External Buildroot layer
- Stores Cruise-specific packages and infrastructure.
- Generally follows Buildroot's recommended format.
- We also occasionally inject dependencies and functions into existing packages from within br2-external
  - Compared with maintaining Cruise-specific patches on Buildroot itself, we find this the lesser evil.

```
.
├── apps
├── br2-external
│   ├── board
│   ├── Config.in
│   ├── configs
│   ├── dependency-fix.mk
│   ├── external.desc
│   ├── external.mk
│   ├── package
│   ├── package-cleanup.mk
│   └── support
├── buildroot
├── buildroot-cache
├── ci
├── docs
├── linux-kernel
├── Makefile
├── makefiles
├── output
├── OWNERS
├── ReadMe.md
├── remote
├── scripts
└── VERSION.txt
```

# Configuration Layering

- We maintain layers (defined as Kconfig include files) for various packages
- The master configuration file for each board is run through the C preprocessor prior to building.
- We generally maintain layers for things like:
  - Architectures
  - Specific SoCs
  - Common peripherals

```
#ifndef __ARCH_AARCH64__
#define __ARCH_AARCH64__

#include "common/config_base.inc"

# Common settings for AArch64 (64-bit ARM) systems

BR2_aarch64=y

BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_ARM_AARCH64=y

#endif // __ARCH_AARCH64__
```

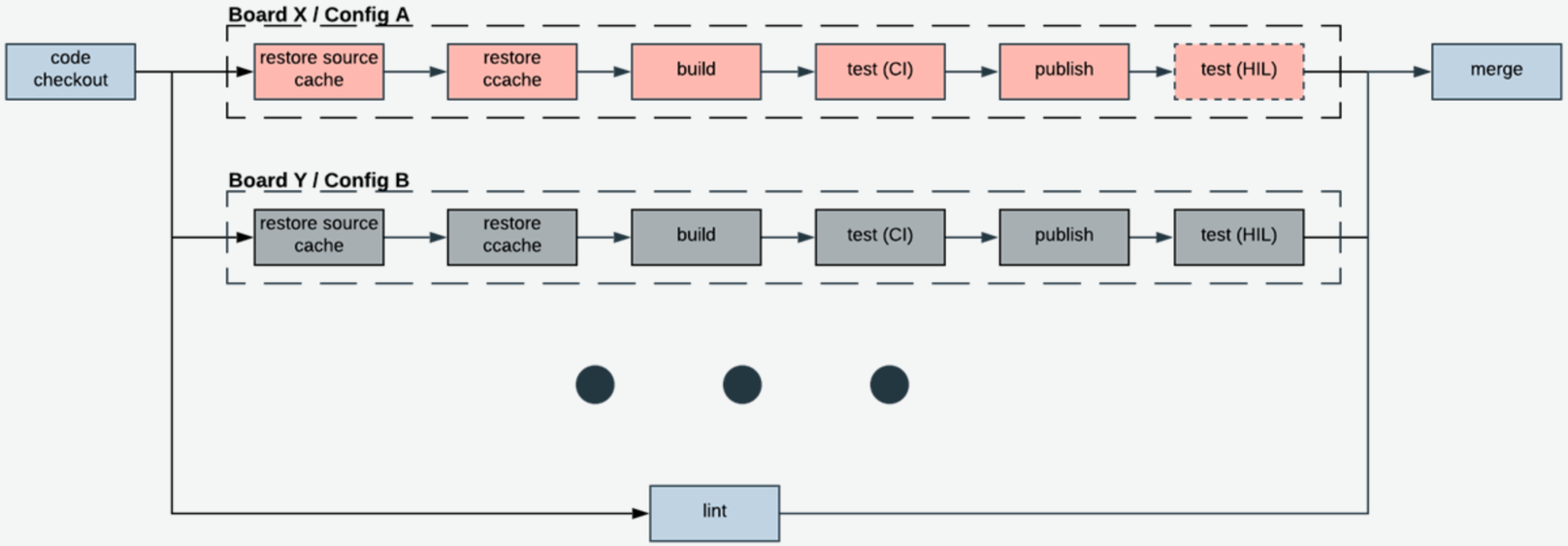
# Testing

- Allow packages to define a function that executes tests.
- Running the new make target '<package>-test' will cause the test to be run
- For now, we've only implemented this for host packages.
  - For some packages, testing the target package in QEMU may be useful but we haven't explored this yet.
  - We'd also like to integrate this with our hardware-in-the-loop (HIL) testing.
- Running '`make host-<package>-test`' will run the host test for that package.
- Running target '`make host-tests`' will cause all tests for configured packages to be run
  - We use this as part of continuous integration (CI).

```
define inner-cruise-test-package

# define sub-target stamps
$(2)_TARGET_TEST =              $$($(2)_DIR)/.stamp_tested

# pre/post-steps hooks
$(2)_PRE_TEST_HOOKS             ?=
$(2)_POST_TEST_HOOKS            ?=

# human-friendly targets and target sequencing
$(1)-test:                      $$($(2)_TARGET_TEST)
ifeq ($(2)_TEST_INSTALL,YES)
$$($(2)_TARGET_TEST):           $$($(2)_TARGET_INSTALL_$(call UPPERCASE,$4))
else
$$($(2)_TARGET_TEST):           $$($(2)_TARGET_BUILD)
endif

# define the PKG variable for all targets, containing the
# uppercase package variable prefix
$$($(2)_TARGET_TEST):           PKG=$(2)

.PHONY: $(1)-test

# Add target to list for global host-cruise-test target
ifdef $(2)_TEST_CMDS
ifeq ($(4),host)
ifneq ($$($$($(2)_KCONFIG_VAR))$$($$($(3)_KCONFIG_VAR)),)
HOST_CRUISE_TEST_TARGETS += $(1)-test
HOST_CRUISE_TEST_BUILD_TARGETS += $(1)
endif
endif # ($(4),host)
endif # $(2)_TEST_CMDS

endef # inner-cruise-test-package
```

# Continuous Integration

- Build & test each each board as a pre-merge check
- Leverage helpful buildroot tooling to speed up builds:
  - Source (download directory) caching
  - ccache support (per board)
- Leverage our test framework to run tests for every *enabled* package that defines them
- Enable automatic testing of the firmware on HIL for applicable systems



# Booting

# Bootloader Consistency

One of our goals is to maintain consistency among platforms.

Bootloaders are inherently board-specific, so consistency is a challenge here.

Still, there are ways...

# General Requirements

First, we need to decide what requirements we can reasonably commonize:

- Secure boot
  - Hardware support for this is a hard requirement at Cruise.
- Redundant OS images
- Initial flashing procedure

# U-Boot

We use U-Boot on our ARM-based boards:

- Well-known and supported
- Extremely configurable

However, this brings some challenges:

- SoC vendors almost always provide a custom fork of U-Boot.
  - May not be recent
  - Feature set can vary based on vendor's interests

# U-Boot

We want to enforce:

1. Easy maintenance: Carry as few patches to the vendor's U-Boot fork as possible.
2. Consistency: Enforce common standards among all boards.
3. Flexibility: Accelerate code reuse while current and future boards may possibly use different U-Boot forks.

How do we do this?

# Common U-Boot Requirements

We can't require all vendors to work from the same tree, but we can enforce requirements as part of procurement:

- Signature validation support
- Network access
  - Useful for bring-up and recovery
- Peripheral access ($I^2C$, SPI, eMMC, etc.)

We can use this to aid part selection, and even if we can't get everything, the gaps are well-known in advance.

# Common U-Boot Functionality

- Use device trees as much as possible.
  - Make this part of SoC vendor requirements if possible
- Use a common U-Boot script to contain common functionality.
  - Scripts are U-Boot best practice
  - Can be signed for security
  - Can be generated from a template to contain common and board-specific functionality.

# U-Boot Scripting

- This is an excerpt of a script where we attempt to netboot as a recovery mechanism.
- By putting code like this into a common script, this functionality is instantly available on all boards.
- We work with our hardware team and any external vendors in the design phase to ensure necessary network access is available in U-Boot.

```
# Try netboot 3 times
setenv i 0
until test $i -eq 3; do
    setexpr i $i + 1
    tftp && bootm
    echo "*** Netboot attempt 0x${i} failed"
    sleep 1
done
echo "+++ Netboot unsuccessful."
```

# U-Boot Changes

Lastly, some U-Boot changes may be unavoidable, like board-specific initialization.

We keep these as minimal as possible, and ideally contained to U-Boot's normal customization methods.

cruise

# Device Management

# A Distributed Computing Environment

Our autonomous driving system is effectively a large distributed computer system.

- Performing an update means deploying software to all nodes in our network.
- All nodes need to be in a known/expected state for the system to function as intended.

# Consistent Upgrades

In this world, consistency is key.  If every component exposes a similar update interface, orchestration of updates is radically simpler and more reliable.

We use swupdate to manage updates among all the devices in our system, using broadly-common configurations.

# Consistent Upgrades

- In order to automate the use of *swupdate*, we provide a REST API to interact with it
- This provides a consistent API across our devices for:
  - Version Querying
  - First-time flash
  - Applying updates
- Building on this REST API allows us to easily build client tools to interact with it in various settings
  - Manufacturing
  - Development
  - Deployment

## swupdate REST API

APIs for interacting with the software update (swupdate) tool

### APIs

#### Get software versions

| Attribute | Value |
|---|---|
| URI | /api/v1.0/swupdate/sw-versions |
| Method | GET |

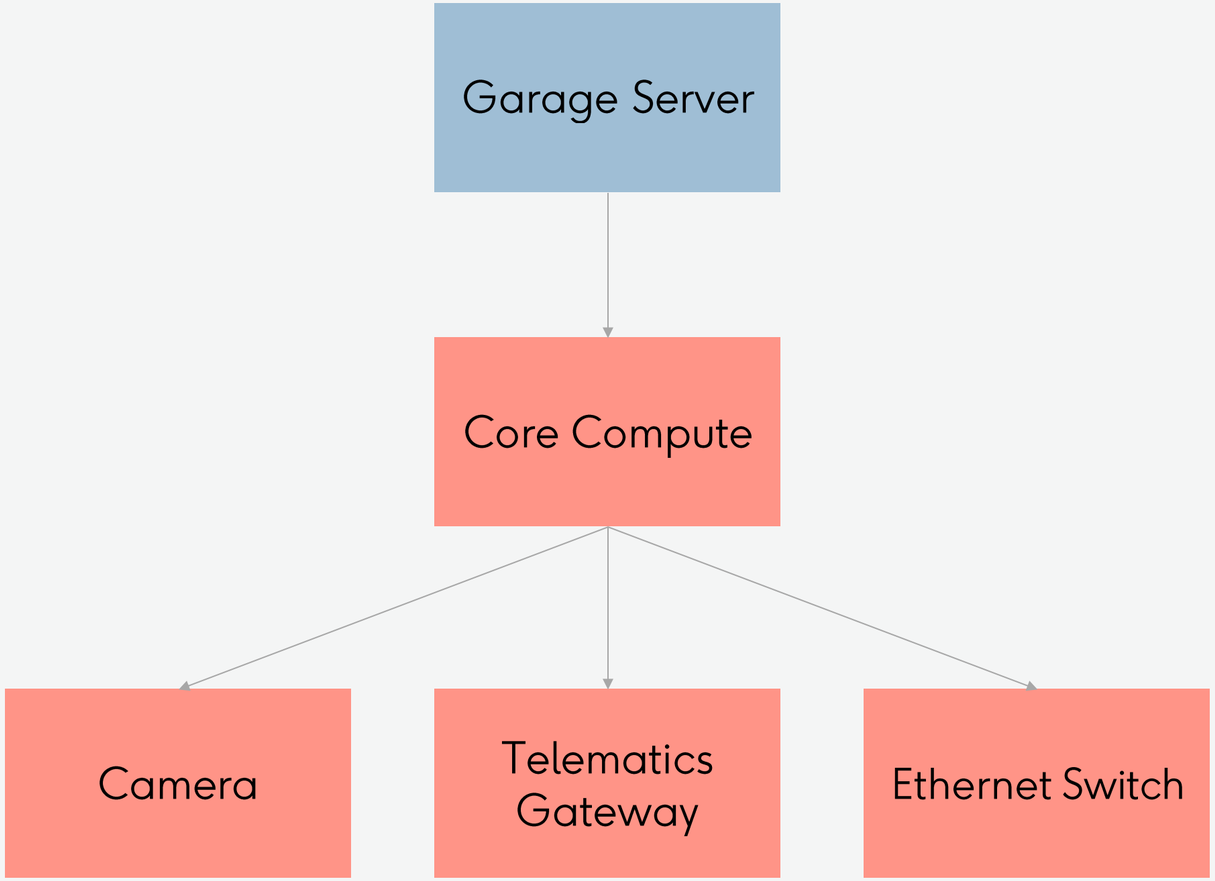
```
# curl http://<url>:<port>/api/v1.0/swupdate/sw-versions
```

Return a JSON message containing the versions of all software running on the system

```
{
    "api": "/api/v1.0/swupdate/sw-versions",
    "status": "success",
    "versions": [
        {
            "name": "<sw pkg 1>",
            "version": "<sw pkg 1 version>"
        },
        ...
    ]
}
```

# Image Deployment

- Core compute functions as master.
  - Can be update itself via swupdate, triggered via a proprietary update service
  - Updates only occur when vehicle is in a quiescent state (i.e. in the garage)
- Core compute can query and update edge devices as needed.
- All devices must be in known and expected states before the AV system can operate.
- It can also function as a TFTP-server-of-last-resort, for devices that enter their TFTP recovery state.



# Redundant Redundancy

Many of our components are difficult to physically access.

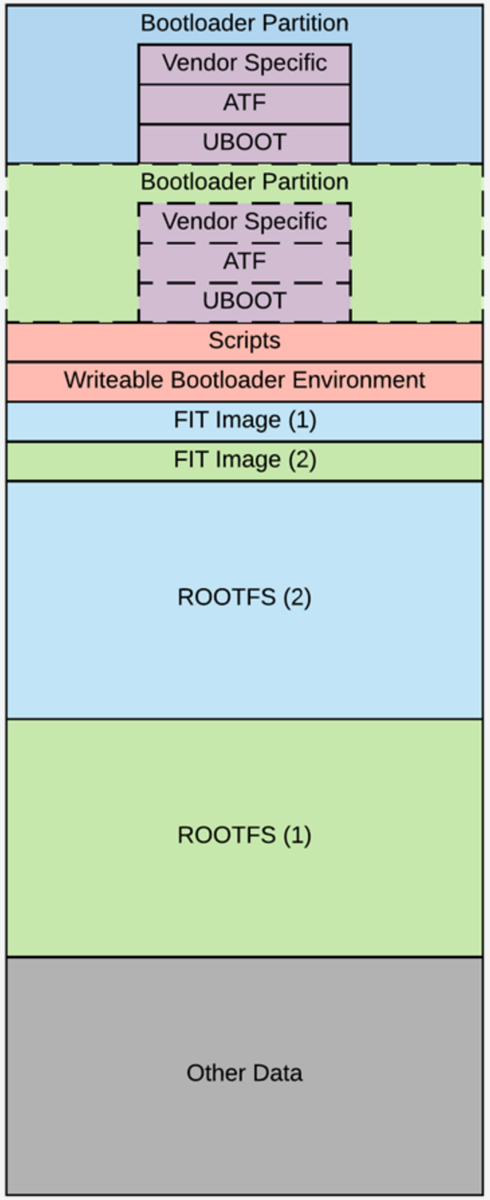
We need to ensure that they're always accessible and are extremely difficult to render inoperative via software.

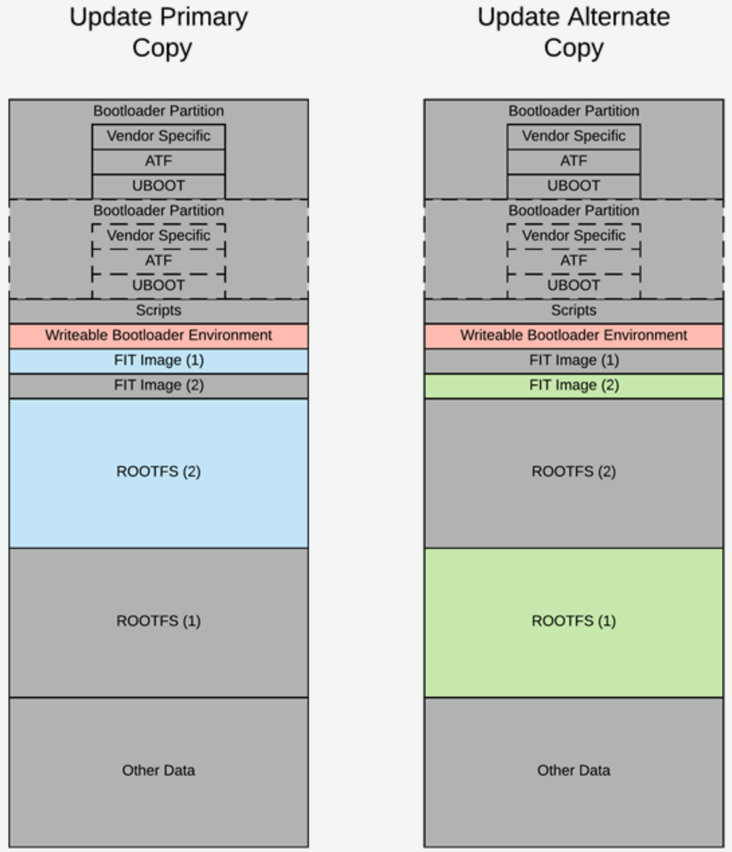
We do this using the common practice of having redundant copies of **everything**.

This includes the bootloader if practical.

# Redundant Redundancy

- Two kernel and rootfs images.
- OS images are secured using U-Boot signed images, rootfs using dm-verity.
- Bootloader automatically attempts to TFTP boot a kernel image if no OS can be loaded.
  - Image still must be signed
  - Also used for initial flash purposes



# Summary

# Putting It All Together

Ultimately, we need to be able to rapidly innovate completely new hardware.

To do that we:

- Use common, well-known, flexible tools
- Define common reusable software components and configurations
- Don’t reinvent the wheel (there’s plenty of other parts of the car to reinvent)

# More Information

- Some relevant Cruise blog posts:
  - Cruise Hardware: How we source and design hardware
  - Vehicle Security: How we think about security
- Other info
  - We were heavily influenced by previous ELC talks, especially this one by Yann Morin at ELCE 2017.

# Thank you