Android* Software Updates

Andrew Boie
Intel Corporation
andrew.p.boie@intel.com
18 March 2015

INTEL OPEN SOURCE TECHNOLOGY CENTER

Agenda

- Introduction
- Filesystem Configuration
- Releasetools
- Recovery Console
- Updater
- SW Update Lifecycle
- Wrap-up

INTRODUCTION

Capabilities

- Full Image SW Updates
  - Updates from any compatible prior SW version
  - At creation time, can set flag to enforce rollback protection; older SW versions cannot be installed on top of newer ones
- Incremental SW Updates
  - `bsdiff / imgdiff` patching on a block-level basis for filesystems boot images
  - Older-style per-file basis patching still supported, but incompatible with dm-verity
  - Incremental updates typically much smaller than a full image update
- AOSP code updates /system, /vendor and boot images
  - Extensible to update special firmware or baseband software via plug-ins
- Update zip files have embedded digital signature
  - verified by android.os.RecoverySystem.verifyPackage() API and also the Recovery Console
- Safely restartable in the event of a power loss
  - Patched files are swapped with originals with rename()
  - Device keeps booting into RC until process is complete
  - RC itself updated after SW update is done and booted into new Android image via oneshot `flash-recovery` *init* service
- At creation time, updates can be configured to force Factory Data Reset in case update has incompatible userdata
  - In a perfect world, only used during development, end users will hate you
- *Updater* program which actually applies the update lives inside the signed SW update package and not on the device

AOSP Software Update Components

- Releasetools
  - Create digitally signed software updates from a target-files-package (TFP)
    - TFP generated by Android* build system
  - Substitute testing for production keys inside a TFP
- android.os.RecoverySystem APIs
  - Framework APIs to verify & install SW updates
  - Handles verification of OTA update digital signature
  - Writes RC command files into /cache/recovery and reboots into Recovery Console
  - Also used to engage Factory Data Reset
- Recovery Console (RC)
  - Alternate boot environment
  - Verify & Apply SW updates
  - Perform Factory Data Reset
  - Typically controlled by command files left by RecoverySystem APIs
  - Hidden menu for manual interaction
- Updater
  - SW update logic, binary inside SW update package
  - AOSP implementation runs script in Edify language
  - Platform-specific tasks implemented in plug-ins
- SW Update UI intent from Settings application

Missing Pieces

- Not all components for an end-to-end solution are open source
  - Remote server backend to host updates (the Server)
  - Client-side mechanism to check and download updates (the Fetcher)
  - Client-side notification UI that SW Updates are available (the Notifier)
- Clearly defined layers of abstraction should make these more or less drop-in
  - Only the server and fetcher care about OTA protocol unless part of a larger device management framework like OMA-DM
  - Fetcher/Notifier in most cases the same APK
  - Works with rest of the SW Update system via android.os.RecoverySystem APIs
  - Settings app integration:
    - Notifier APK should have activity with intent filter for android.settings.SYSTEM_UPDATE_SETTINGS to check for updates
      - Launched when user clicks About tablet -> System Updates
  - BootReceiver to check for RC messages in /cache/recovery
  - No other framework modifications should be necessary
    - Any vendor that proposes invasively hacking up the framework to support OTA should be regarded with extreme suspicion unless they are doing OMA-DM

Limitations

- No support for disk re-partitioning
  - Leave lots of slack space in `/system` for future Android releases
  - Leave slack space in `boot` and `recovery` partitions for future boot images
- One update applied at a time
- Device can’t be used while updates are applied
- Errors during update typically require RMA

FILESYSTEM CONFIGURATION

Partition Layout

- `boot`
  - AOSP boot image
  - Linux kernel and ramdisk mounted as root filesystem
  - Contains *init* and essential tools to mount `/system` and boot the rest of Android
- `system`
  - All Android system applications and libraries
  - Always mounted read-only
    - Incremental updates depend on `/system` being in a specific state
      - For ext4 with dm-verity, down to the block level
      - Otherwise, on a per-file basis (old way)
    - Mounting read-write at any time modifies the ext4 superblock
- `vendor`
  - Found on some Android devices, similar purpose and limitations as /system
  - May contain vendor-specific apps and libraries not provided by AOSP
- `oem`
  - Found on some Android devices, contains vendor-specific oem.prop file, branding, bundled apps
  - If oem.prop file affects OTA update, pass it to `ota_from_target_files` with -o option
  - Not touched by OTA updates, if you put apps on there they can only be updated via Play Store or equivalent
- `data`
  - Downloaded applications
  - Application data
  - Dalvik cache
  - Typically not touched by OTA updates
  - Erased by Factory Data Reset

Partition Layout (continued)

- `recovery`
  - Alternate AOSP boot image
  - Ramdisk contains ‘recovery’ program which is the RC implementation
- `misc`
  - Very small size, does not contain a filesystem
    - Has Bootloader Control Block (BCB) written directly to the block device node
  - Used to communicate between RC and the bootloader
- `persistent`
  - Cached user credentials for untrusted Factory Data Reset
  - Flag to allow "Fastboot OEM unlock" stored here
  - Not touched by the OTA system
- `metadata`
  - Contains crypto metadata for encrypted userdata
  - Not touched by the OTA system
- `cache`
  - Temporary storage, contents can be erased at any time
  - Used by some applications as a temporary storage/download area
    - Requires special APK permissions
  - Downloaded OTA updates used to be stored here
    - Recovery Console can now read OTA update data directly from userdata partition
    - pre-recovery service un-encrypts blocks containing update file before launching into RC
  - Used as temporary storage by *applypatch*
  - Erased during Factory Data Reset
  - CDD specifies 100M size, used to be 2/3 size of /system

Android Boot Images

- Container file format, with metadata, kernel image, ramdisk, and optional 2nd-stage bootloader image
- Created by `system/core/mkbootimg`
- Used in two places
  - Called by the build system to create boot images in `$OUT` as part of a normal build
  - Also created by *Releasetools* when assembling an OTA update from a target-files-package
- After creation by `mkbootimg`, signed with `boot_signer` utility
  - `system/extras/verity`
  - Part of Lollipop Verified Boot feature

fstab

- Specification file for all the **filesystems** on the device
  - Not a partitioning specification, no offset/ordering/partition size information
- Used by RC, fs_mgr, and *Releasetools*
- In older versions of Android
  - Used to be a "recovery.fstab" file only used by RC
  - Mounting of images at boot was done by manual mount commands in init.rc
  - Nowadays mounting done by fs_mgr, shares fstab format with RC
- Maps mount points to device nodes and filesystem types

Software Update Creation

RELEASETOOLS

Target-Files-Package (TFP)

- Created by '`make target-files-package`'
  - Build logic in `build/core/Makefile`
- Zip file containing snapshot of a particular SW release, everything needed to create an OTA update
  - A single TFP used to create a full image update
  - Two TFPs used to create an incremental update
- Subdirectories containing image contents
  - BOOT/, RECOVERY/, SYSTEM/, VENDOR/
  - Prebuilt boot and filesystem images under IMAGES/
- Add additional device-specific blobs by defining "radio files"
  - Seems to be a legacy name, doesn't necessarily have anything to with device's radio
  - In AndroidBoard.mk:
    - `$(call add-radio-file,myblob.dat)`
  - Ends up in target-files-package in `RADIO/` directory
  - Platform-specific extensions to *Releasetools* will handle these files

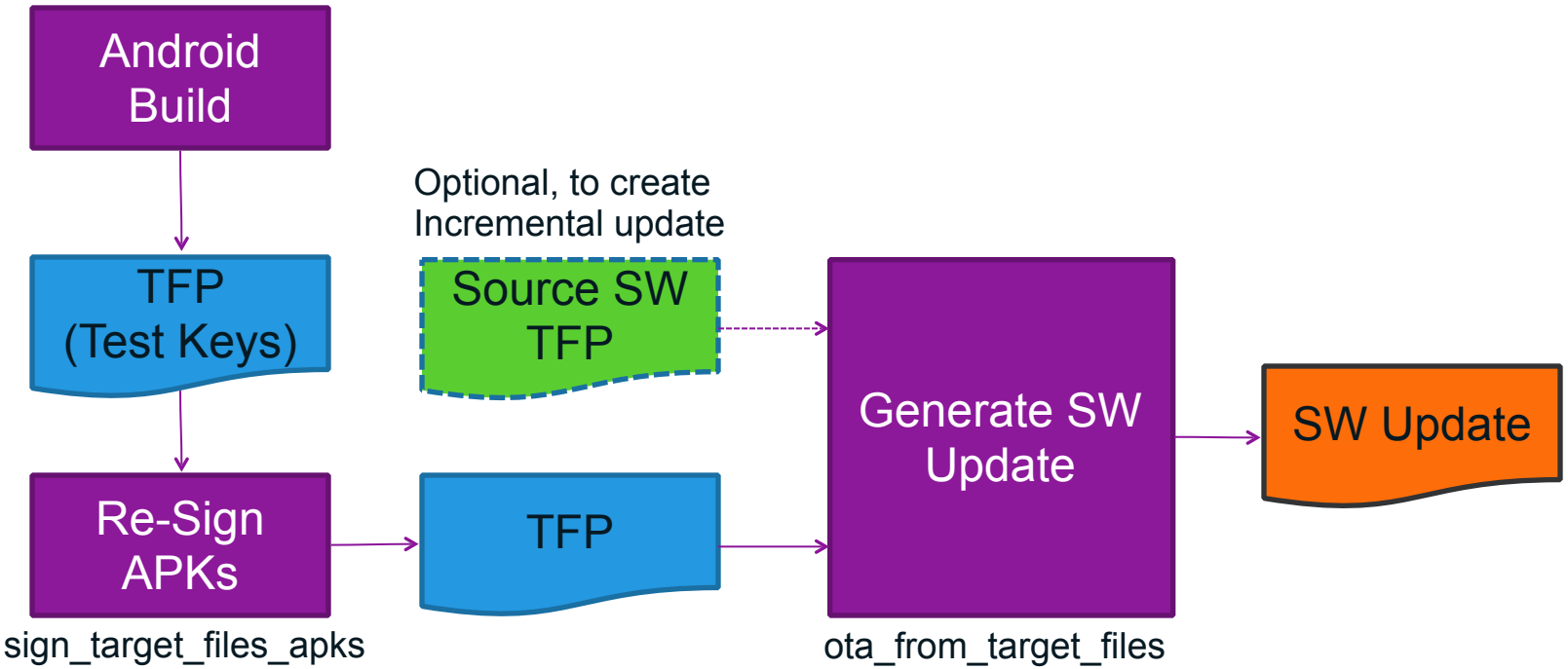
Android Security

- All APKs are digitally signed
  - Updates to the same application must be signed by the same key otherwise app data will be inaccessible
  - Apps that want to share the same user ID must be signed with the same key
  - LOCAL_CERTIFICATE specifies the key; defaults to **testkey**
- OTA updates will be discarded if not signed with one of the expected keys
- AOSP has five keys in build/target/product/security
  - Testkey: Default key to sign APKs
  - Platform: Core platform packages are signed with this
  - Shared: Data sharing between Home and Contacts processes
  - Media: Packages in the media/download system
  - Verity: Signed boot & recovery images, dm-verity root hash for /system and /vendor
- 2048-bit RSA keys with public exponent 3. Created with *openssl*
- Cannot ship a product with the AOSP keys that are in the build
  - Test keys should only be used during development, enforced by CTS case
  - `sign_target_files_apks` tool in *Releasetools* used to swap out the AOSP test keys in the TFP with the real production keys
    - Re-signs all APK files with new keys
    - Recreates boot, system, recovery, vendor images signed with new verity key
    - Swaps out OTA verification key in RC ramdisk
    - Swaps out dm-verity root hash key in boot ramdisk
    - Prebuilt images under IMAGES/ updated

ota_from_target_files

- Lives with the rest of the *Releasetools* in `build/tools/releasetools`
- Script to create OTA updates from one or more TFPs. Some useful options:
  - `--incremental-from <TFP>`
    - creates an incremental update from a source TFP
  - `--wipe-user-data`
    - OTA package will erase /data when installed
  - `--no_prereq`
    - Disable rollback protection checks
  - `--package_key`
    - Key to use to sign the package; defaults to testkey
  - `--block`
    - Generate block-level OTA updates
  - `--two-step`
    - Recovery Console is updated first, then other update tasks are run
    - Typically not needed unless updater inside OTA package has dependency on new kernel
- For most platform-specific update tasks, Python extensions can be implemented to perform additional tasks in the update
- Creates a SW update package containing all images, *updater* binary, and Edify script that *updater* interprets to apply the SW update

SW Update Creation



Releasetools Extensions

- "Radio files" specified in AndroidBoard.mk will be populated in the TFP
- To extend *Releasetools* to work with these files, create a Python module with the following functions
  - `FullOTA_Assertions()`
    - Called after emitting the block of assertions at the top of a full OTA package
  - `FullOTA_InstallBegin() & FullOTA_InstallEnd()`
    - Called at the end of full OTA installation
  - `IncrementalOTA_Assertions()`
    - Called after emitting the block of assertions at the top of an incremental OTA package
  - `IncrementalOTA_VerifyBegin() & IncrementalOTA_VerifyEnd()`
    - Called at the beginning/end of the verification phase of incremental OTA installation; put checks here to abort the script before any changes are made
  - `IncrementalOTA_InstallBegin() & IncrementalOTA_InstallEnd()`
    - Called at the beginning/end of incremental OTA installation
  - All functions passed an 'info' option which has pointers to
    - `zipfile.ZipFile` objects for source and target TFPs, output zip file. Used to move files or create patches from TFPs and place inside the SW update
    - Script object to append additional Edify script commands
    - Use `ota_from_target_files` code as a guide
- Put path to this module in `TARGET_RELEASETOOLS_EXTENSIONS` in BoardConfig.mk
- Relevant `ota_from_target_files` options
  - `--device_specific` path to Releasetools extension module
  - `--extra` to pass additional key/value pairs accessible by *Releasetools* extension
  - `--extra_script` to directly add commands to the generated *updater* Edify script

Other Releasetools

- `img_from_target_files`
  - TFP as input
  - Produces an image zipfile suitable for use with '`fastboot update`'
  - Just flashes prebuilt images in IMAGES/ subdir
    - boot, recovery, system, etc
- `check_target_files_signatures`
  - Looks for problems with package signatures inside a TFP
  - Can be used to check for compatibility problems between two TFPs (key changes, etc.)
- `add_img_to_target_files`
  - Used to update the IMAGES/ directory in the TFP
- `build_image.py`
  - Code to create signed ext4 filesystems with dm-verity metadata and hash tree
- `make_recovery_patch`
  - Build system tool to create the recovery.img patch from boot.img so that it can be populated in the system image
  - Recovery patch can't be inserted into system image after it is made due to signing, so generation moved much earlier in the build process

SW Update Alternate Boot Environment

RECOVERY CONSOLE (RC)

Recovery Console

- Alternate boot image, a few ways to enter
  - `reboot recovery` from a shell
  - `RecoverySystem` APIs in Android Framework
  - OEMs sometimes implement a bootloader ‘magic key’
- Pictorial interface
  - Recent versions of Recovery Console now allow for multiple screen resolutions and localized text strings during update process
    - Only 4 strings (error, installing, no_command, erasing), rendered as image files
- Hidden non-localized menu for manual tasks
  - Factory Data Reset
  - "Sideload" OTA update over ADB
  - Other platform-specific tasks as implemented in Recovery Console UI plug-in
    - SD Card installation of OTA package no longer part of default UI
- Log files saved in /cache/recovery
  - All stdout/stderr from RC and *Updater*
  - Edify ui_print() commands
  - Log files can now be directly viewed in the Recovery Console UI

/misc and the BCB

- Tiny partition used for communication between *RC* and bootloader, and for RC to save state information
- Contains **Bootloader Control Block** (BCB)
  - `command[32]`: Commands for the bootloader
    - “boot-recovery” boot into RC instead of Android
    - Other platform-specific commands may be implemented for update tasks that must be done by the bootloader
    - If empty, garbage, or no known commands matched, normal Android boot
  - `status[32]`: Return status field written by bootloader after performing platform-specific commands
    - No specification, platform-dependent
  - `recovery[1024]`: Command line for Recovery Console
    - Arguments tokenized by ‘\n’
    - Invalid if first argument not ‘recovery’

Recovery Console Control

- Upon startup, looks for command line arguments in decreasing precedence:
  - Actual command line to 'recovery', debug-only scenario
  - BCB.recovery
  - Command file in /cache/recovery/command (written by RecoverySystem APIs)
    - RecoverySystem doesn't write to BCB due to permissions on doing raw block device I/O
- Always copies arguments into BCB.recovery and sets BCB.command to "boot-recovery"
  - Makes sure we keep booting into RC with the same arguments in event of unexpected power loss
  - Don't rely solely on /cache/recovery/command for this
- `finish_recovery()`
  - Called when requested operations (SW update, factory data reset, etc) are complete, whether successful or failed
  - BCB is cleared so that subsequent reboot goes back into Android
  - Rotates log files under `/cache/recovery/`
  - If no arguments were given to RC, displays error image and waits for menu input
- A divergent update process is **very very bad**, should always at some point complete so that `finish_recovery()` can be called
  - Else the device will get stuck, user resets it, gets stuck again, can never boot back into Android

Bootloader Integration

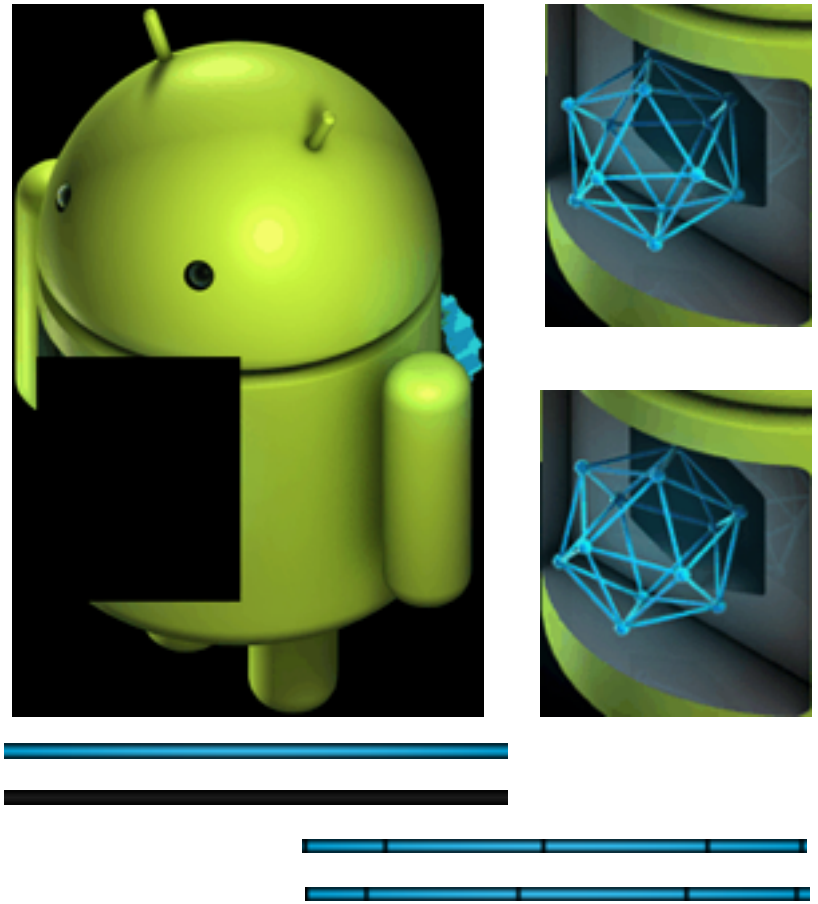
- Linux kernel should do a one-shot boot into Recovery Console boot image if "`recovery`" is supplied as a reboot() argument
  - Implement in a driver via `register_reboot_notifier()`
  - Communicating this to the bootloader platform-specific
    - On EFI devices we used an EFI Variable
- Bootloader selects boot image (or other task) at boot
  - Check BCB.command field
    - Set by RC when it first begins an update
    - BCB.command is persistent; keep booting into RC until RC clears it
    - Garbage or zeroed out contents should simply boot into Android
  - Check platform-specific mechanism for one-shot boots

Recovery Console UI Plug-in

- Device-specific policy and extensions for RC
  - Copy bootable/recovery/default_device.cpp into your $(TARGET_DEVICE_DIR) space and create as a static library
  - Set TARGET_RECOVERY_UI_LIB in BoardConfig.mk to the $(LOCAL_MODULE) for this library
- Capabilities
  - Implements `Device` class, see bootable/recovery/device.h
  - Define additional menu items in `Device::GetMenuHeaders()` `Device::GetMenuItems()`
  - Extra device wiping code implemented in `Device::WipeData()`
  - Customization of branding graphics done in `Device::GetUI()`
  - Additional initialization tasks in `Device::StartRecovery()`
  - Customize effect of keystrokes with Device::HandleMenuKey`()`
    - Key chords can be implemented by additionally checking `ui_key_pressed(code)`
  - Implementation of device-specific commands in `Device::InvokeMenuItem()`

Recovery Branding

- Image-only GUI; in case of a problem user must call customer care
- Built-in images can be replaced by putting images in `$(TARGET_DEVICE_DIR)/recovery/res`
  - Must be 8-bit PNGs in RGB or RGBA format
  - Keep same filenames
- Number of animation frames for items, offsets, etc., can be set in UIParameters struct passed to `device_ui_init()` in UI plug-in
- `bootable/recovery/make-overlay.py` script helps create the base image plus overlay frames and computed offsets for animations



UPDATER

Updater binary

- Lives inside the SW update zip
  - No need for security checks since signature of SW update has been verified
  - Nice to have this in the zip so that all SW update implementation code doesn't need to exist a priori
  - If certain update tasks require an updated kernel, use 2-step updates which reboot into an updated Recovery Console before continuing
- Not strictly required to use the *updater* implementation in AOSP
  - RC fork/execs *updater* and communicates via pipe
  - RC passes in 3 arguments
    - RC API version (currently 3)
    - file descriptor for communication pipe
    - path to the SW update zip file
  - *Updater* writes string commands over the pipe to set progress bar parameters or print strings to the hidden Recovery UI; see `bootable/recovery/install.c`
- AOSP implementation interprets a script inside the SW update written in Edify language
- Platform-specific *updater* capabilities implemented in plug-ins
- **Be sure *updater* terminates**! Corner cases where it doesn't terminate effectively bricks the device

Recovery From Boot

- RC itself is not updated during the course of a SW update
  - Makes sure that the same RC (from the source SW version) is used throughout the update even if power lost
  - Useful for key revocation scenarios
- Oneshot init.rc service `flash_recovery`
  - Runs `/system/etc/install-recovery`
  - Checks SHA1 of Recovery image to detect whether it needs to be patched
  - No-op if matches patched SHA1
  - Applies a patch to the boot image to create the RC image, which is written to the recovery partition

Applypatch

- Used by incremental updates to patch /system files and boot images
- When calculating binary diffs, is able to detect compressed file headers and break files into chunks
  - For gzip, does not work well unless file was compressed with same version of deflate() algorithm that it has access to
  - ‘gzip’ has a much older deflate() than zlib (gzip does not link against zlib, it is self-contained)
  - Hence all ramdisks should be compressed with zlib-linked ‘minigzip’ and so should bzImages
    - May have to hack your kernel build to use minigzip instead of gzip
    - I sent a patch to LKML to add minigzip as a supported compression type in Kconfig but the response was unkind
- Uses bsdiff algorithm
  - Bsdiffs can take a while to compute
  - In particular, diffs of files that both contain large regions of all 0s elicits worst-case $O(n^2)$ performance – don’t diff padding!

Edify Language

- Scripting language to specify SW update tasks
- Everything is an *expression*
  - Expressions evaluate to strings
  - ; operator is a sequence point, value returned is right side
  - Boolean operators supported, concatenation, equality
  - `if … then … else … endif` and `if … then … endif` blocks
  - Empty string is Boolean “false”, all other strings are “true”
  - Functions return expressions and take expressions are arguments
  - No language support for loops
  - Short-circuiting && and || operators
- All functions implemented in C, cannot declare functions in an Edify script
- Implementations for built-in Edify functions related to OTA in bootalble/ recovery/updater/install.c
  - Use this code as a guide when implementing your own functions
- Additional functions implemented in plug-ins\
- See `bootable/recovery/edify/README` for language specification

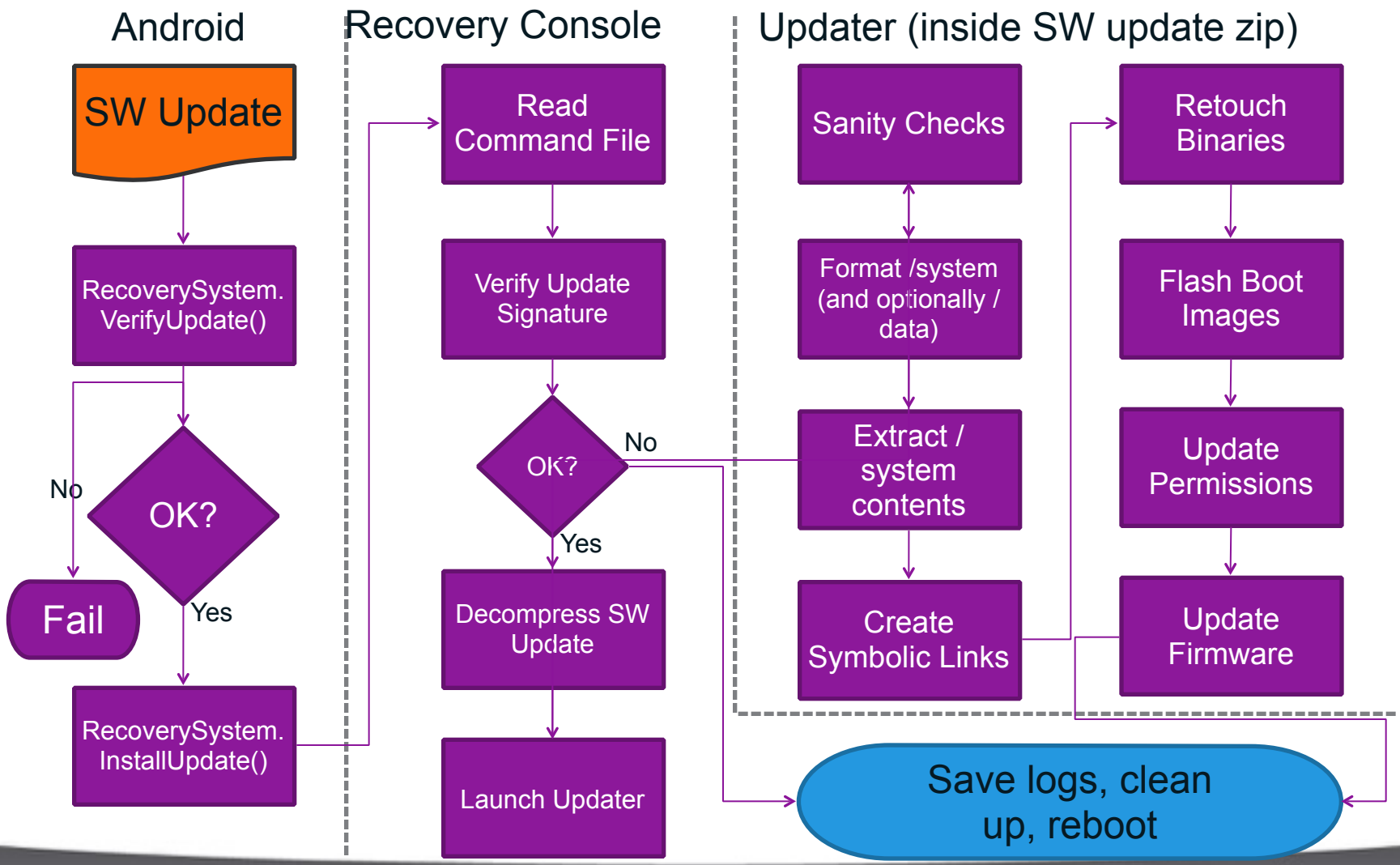
Updater Plug-ins

- Multiple updater libraries can be defined
  - Create a static library in Android build in device-specific area
  - Put module names in BoardConfig.mk `TARGET_RECOVERY_UPDATER_LIBS`
  - Each library needs a registration function, which gets called by *Updater* when it starts up
    - `void Register_$LOCAL_MODULE()`
    - Calls `RegisterFunction()` for each new Edify command
  - Use `bootable/recovery/updater/install.c` as a guide
  - Additional supporting static libraries can be added to updater by putting their module names in `TARGET_RECOVERY_UPDATER_EXTRA_LIBS`
- Edify API defined in `bootable/recovery/edify/expr.h`

Putting It All Together

SW UPDATE LIFECYCLE

SW Update Application (Full Image)



WRAP-UP

Productization Tasks

- Define fstab file and point TARGET_RECOVERY_FSTAB to it
- Add platform-specific blobs via “radio files” Makefile directives
- Implement RC UI plug-in (optional, default one sufficient for many)
  - Add branding images if necessary
  - Add platform-specifc tasks
  - Map the buttons how you want them
  - Additional tasks for Factory Data Reset
- Implement *Updater* plug-in
  - Implement Edify commands in C for platform-specific update tasks
- Implement *Releasetools* extensions
  - Add logic to patch or add RADIO/ images to the OTA update
  - Add Edify commands to handle them in updater script
  - Add additional assertions/verification steps as needed
- Add BoardConfig.mk variables to declare all plugins/extensions and supporting libraries
- Hook up download agent to Settings application via intent filter
- Bootloader integration to use BCB
- Kernel/bootloader integration for one-shot boot targets
- Generate production keys
- Shouldn’t need to modify the Android framework, build/core, or bootable/recovery
- Have a GOOD ongoing test plan and dogfood as much as possible

Tips

- Firmware updating
  - If FW image can be read back at runtime:
    - Read FW image and save to a file
    - Apply binary patch with applypatch
    - Flash it back
  - Otherwise:
    - Put the firmware image somewhere on /system
    - Will get patched with everything else on /system during the update
    - Query current FW version, compare to version living on /system
    - If different flash it
  - For FW that affects device boot, make sure FW flashing is safely restartable!
- You should not have to modify the RC, do all your extensions in plug-ins
  - Rule of thumb for all Android platform development: don't change the framework unless you have a really good reason to do so
  - If you do modify Android, do so with upstreaming in mind
- Incremental updates assume /cache is mounted
  - No explicit guarantees that RC will mount it for you
  - Seems to be a bug in ota_from_target_files, add script.Mount("/cache") right before script.CacheFreeSpaceCheck()
- Testing, testing, testing!!!!
  - Create SW update packages in your internal releases and make people (field testers, QA, developers) use them
  - Automated testing is also your friend
    - Buildbot, Jenkins, etc
  - Software update is the one mechanism on the device that *has* to work
    - Fortunately *Updater* is in the SW update package and not on the device

Legal Disclaimer

INTEL OPEN SOURCE
TECHNOLOGY CENTER