Apertis System Update and Rollback Design

Author:	Gustavo Noronha
Contributors:	Sjoerd Simons, Derek Foreman
Version:	v1.6.3
Status:	Final
Date:	16 November 2015
Last Reviewer:	Luis Araujo

DOCUMENT CHANGE LOG

Version	Date	Changes
1.6.3	2015-11-16	• Update to new name Apertis • Remove file custom properties (metadata)
1.6.2	2014-12-15	• Updated to new template
1.6.1	2013-04-15	• Update chapter 8 to reflect actual implementation
1.6.0	2013-03-13	• Rename second listing 2 to listing 3 • Better explanation of intent to use snapshotting mechanisms for atomic operations at the start of chapter 5.1 • Add explanation of splitting large files before archiving to chapter 8 • Add Illustration J: Split File Extraction and details of the archival and extraction process to chapter 8
1.5.0	2012-12-03	• Update section on effects of system rollback on applications to be in synch with current thinking in the Applications Design • Explain the kernel upgrade process for component updates • Add a chapter on file archiving • Explain decision to use initiral ramdisks • Add caveats to suggestion of LVM as a BTRFS fall-back • Rename “Chunk” files to “piece” files for consistency with older text • Remove duplicated text from Applications Design, replace it with high level overviews of application concepts.
1.4.2	2012-11-06	• Address internal reviewer comments • Add a brief synopsis of flash translation layers in 2.3.1 Flash Translation Layers (FTLs) • Mention zRAM in 2.7 Swap Space • Explain file splitting problem and potential solutions for split updates
1.4.1	2012-10-19	• Add a chapter on settings migration • Synch application section with Applications Design • Explain where lock-files should go, as well as adding some explanation of the various mount points in 2.1 File-systems and layout • Retract recommendation against initrd • Explain tmpfs memory requirements • Add section on boot flag robustness/layout • Clarify situations when factory images may not be bootable

		• Reference application update information from 3 Update management • Minor clarifications to 3.6 The Post-Update Restart • Discuss the possibility of swap space • Move and expand text on GPT flags (2.3) • Move backup flags to end of disk • Add a new flag to track updates
1.4.0	2012-07-27	• Add factory fallback image • Define flags • Rename userspace stages • More details about storage • Revised boot and rollback procedures • Integrate contents of sysgo documents • Rename stage-1 and stage-2 to Mini Userspace and Full Userspace
1.3.2	2012-05-15	• Address review comments
1.3.1	2012-05-11	• Updated title and filename to Document Naming Scheme
1.3	2012-05-08	• Add section on application storage • re-order sections
1.2	2012-04-17	• Update to reflect lack of NAND storage • Multi stage userspace • Storage layout changes
1.1	2012-04-03	• Made it clear that the clean slate – unpack image upgrade method is an opt-in • Added section on upgrades from external media
1.0	2012-03-14	• Final tidy up
0.5.0	2012-03-12	• More details on the LVM solution • User data section
0.4.0	2012-03-08	• Addressing review comments
0.3.0	2012-03-07	• Software management moved to API document • Improvements to the boot up discussion, with new diagrams to help describe it
0.2.0	2012-03-01	• Software management chapter
0.1.0	2012-02-06	• Initial release

Table of Contents

Illustration Index

1 INTRODUCTION

This document is intended as a discussion starter for an assorted collection of topics. Currently these topics are:

- System boot
- File-system layout
- Update and rollback infrastructure
- User and application storage

While this document discusses how core software and applications are laid out on storage, detailed discussion regarding their interfacing with the system is the subject of the Supported APIs document instead.

User storage requirements are fully defined in the Multi-User Design, and applications are the subject of the Applications design. These topics are only briefly mentioned in this document.

2 PARTITIONING AND BOOT

The boot sequence uses a set of flags to determine which kernel and root file-system (snapshot) to use. As the flags are set based on the success or failure of previous boots and upgrade, this allows for a system that can rollback to an older system snapshot and fallback to the older kernel as necessary.

For partition management the GUID Partition Table (GPT)[1] system was chosen for being flexible while having fail-safe measures, like keeping checksums of the partition layout and providing some redundancy in case errors are detected.

Illustration A: Storage layout shows how the storage space would be divided for the various uses. The NOR flash device, (called Memory Technology Devices - MTD in Linux) would store the boot loader and its configuration. The bigger flash storage will hold any fallback images, the system volume to be mounted read-only, with a snapshot or reserved space. Finally, there is the general storage volume, where system configuration lives and applications would also be installed. Unless an upgrade is in progress, the only volume with write access will be the general storage area.

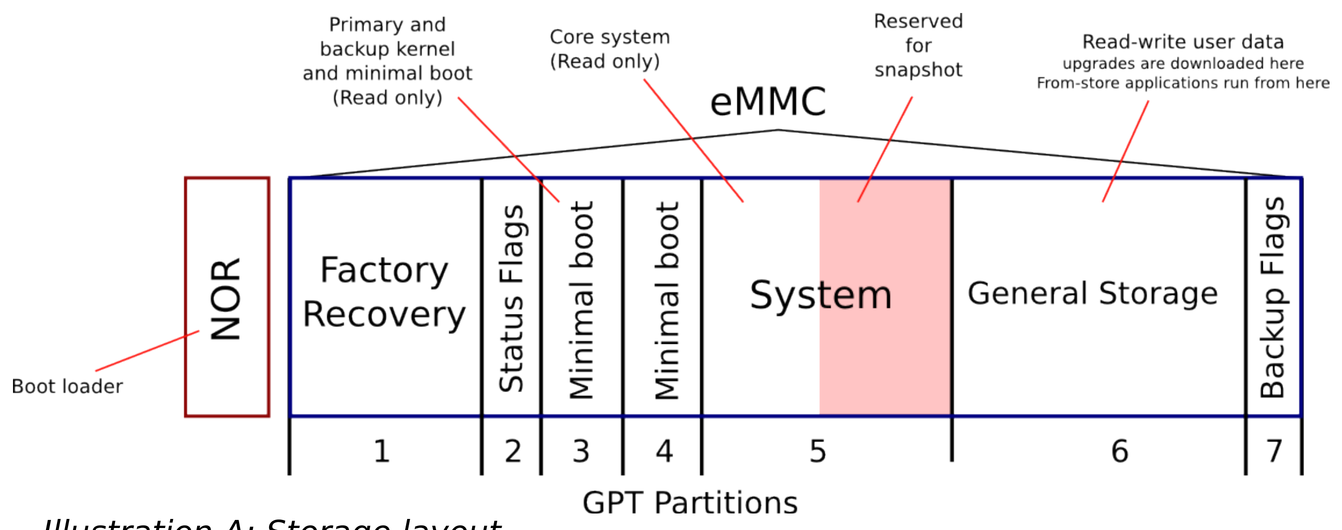


Illustration A: Storage layout

NOR is usually used as a replacement for simpler EEPROM chips, and is the more expensive part of this setup. It's the ideal place for the boot loader to live. The NOR is wired to the CPU, and can be used as part of the system memory, so code can be run directly from it, without being loaded onto RAM first, which may be a good way of saving RAM, but the performance impact of this approach would need to be assessed in real world hardware before it is adopted.

Collabora is currently considering a multi-stage boot process (Illustration B: Boot process overview) in which a quickly booting “Mini Userspace” will perform time sensitive startup tasks before booting a “Full Userspace” to provide complete

1 http://en.wikipedia.org/wiki/GUID_Partition_Table

system functionality.

The “Mini Userspace” will be an initial ramdisk[2] stored in a one of the “minimal boot” partitions on the device along with the kernel image required to run it, and any other mandatory boot loader configuration files.

Using the Linux initial ramdisk functionality incurs some extra overhead at boot time – the ramdisk image file must be loaded and decompressed into memory before it can be used. However, this delay should only be on the order of a few milliseconds.

Significant development time savings can be had in using the existing initial ramdisk configuration from the Ubuntu distribution as a starting point, but if Apertis boot time is too long, this is one area where some small savings may be had, and the decision to use an initial ramdisk instead of a filesystem on a partition can be revisited.

Please note that details of the secure boot process are currently missing from this document. It's important for a system of this nature to ensure that it's only running trusted code, from the boot loader to userspace. The exact details of the secure boot system are currently unknown, and will be provided in a future revision of this document.

2 An initial ramdisk is a small filesystem image loaded by the kernel at boot to perform setup tasks before the root filesystem is mounted.

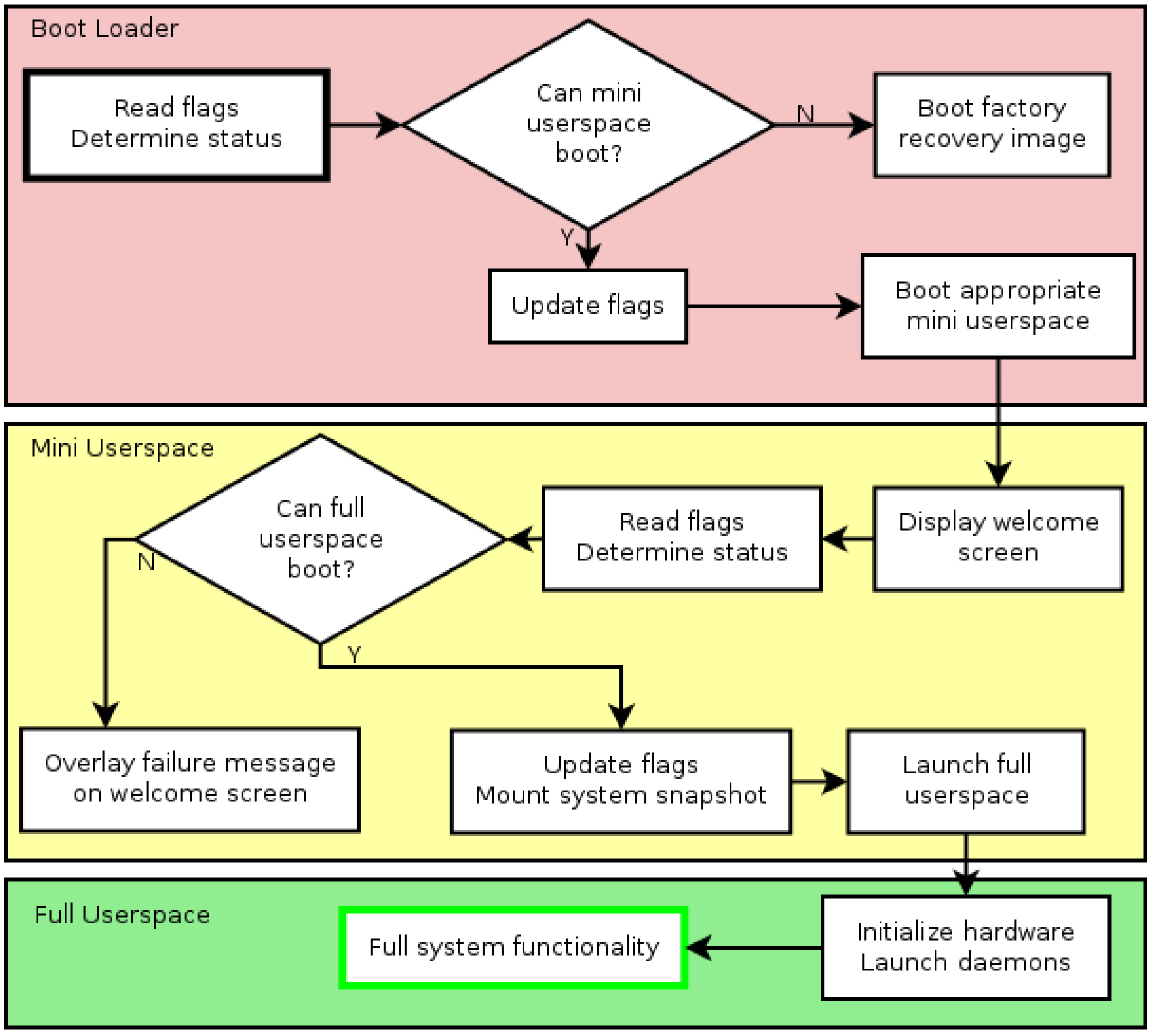


Illustration B: Boot process overview

2.1 FILE-SYSTEMS AND LAYOUT

The flag partitions will store binary data directly and have no need for a filesystem at all. The bulk of the storage device (system, general storage) will use BTRFS[3]. However, BTRFS is not an optimal choice for the remaining partitions in the system.

The factory recovery and minimal boot partitions can't use BTRFS because the boot loader is incapable of booting a kernel off of a BTRFS filesystem. These partitions will have no use for the advanced features of BTRFS, and they will be read-only with no need for a journalling filesystem. Either ext2fs or FAT32 would be good choices for the factory recovery and minimal boot partitions. Historically, the chosen boot loader has been more stable when using FAT32 than ext2, so FAT32 has been selected.

3 4.1 The case for and against BTRFS provides the reasons for this decision.

For the most part, the system will follow the Filesystem Hierarchy Standard: http://www.pathname.com/fhs/pub/fhs-2.3.html

though an additional directory will be added for application storage. (See the Applications Design).

The general storage area will be split into subvolumes to be mounted on:

- /home
- /var
- /Applications

The storage requirements of the /var will be on the order of a few tens of megabytes, but the /home and /Applications directories will depend entirely on the users of the system.

It is intended that the bulk of user data storage be stored along-side the applications in the /Applications hierarchy and as such be governed by the application rollback system as laid out in the Applications Design.

The /home hierarchy exists in the case that per-user storage outside of the rollback system is required. Since /home is the default storage location for many existing Linux applications, attention will have to be paid to its contents during system testing to ensure it is not being used inappropriately.

The minimal boot partition should be as small as possible. For reference, the 2.6.35 kernel image shipped with HP's Touchpad is 3.2MB in size, uncompressed, and boots a userspace weighing around 6MB. Right now, our compressed kernel is around 4MB and our compressed minimal userspace is only around 2MB – but these numbers are preliminary and expected to change.

The size of the full system partition is hard to predict at this point, and is affected by the number of pre-installed applications integrated with the system. Current development images use around 850 Megabytes for system storage[4] (which would require an additional 850 Megabytes in reserve for rollback). The fallback image would take around 300 Megabytes for a system this size. It is probable that final images will be less than half this size.

Small temporary storage will be provided by tmpfs, a ram backed virtual filesystem, and as such will be automatically lost over reboot. More persistent temporary files such as application caches will be in the general storage area as described in the Applications Design. System components needing long term storage can use /var, and manage this space themselves.

Since tmpfs grows automatically to meet file storage requirements, and reduces available memory in doing so, its usage will be limited to lockfiles (such as those stored in /run) and other tiny ephemeral data, and it should consume less than a megabyte of memory. Maximum usage limits will be set to prevent tmpfs from consuming an unreasonable amount of device memory.

Care should be taken to place all lock-files in the /run file heirarchy so they

4 These image sizes are preliminary – no significant effort has yet been made to reduce storage requirements, nor is a full complement of software installed.

automatically disappear on a system restart.

2.2 BOOT LOADER

The go-to bootloader for ARM is U-Boot[5]. U-Boot is scriptable[6] and very modular – it can read and write to flash devices[7], and can initialize additional modules on demand for reading files and settings off devices such as eMMC.

U-boot will be extended to check the power failure and watchdog reset cpu status information, as well as the success/failure statistics from previous boots from the state partition. It will use this information to select a minimal boot partition to boot a Mini Userspace from. The boot attempt will be logged in the status partition so subsequent boots can tell that a failure occurred if the boot doesn't succeed.

U-boot is capable of displaying a splash screen to hide a slow boot – instead of using this feature, we will re-purpose it for critical failure messages. In the event of an unbootable system, it will display a static failure message informing the owner that Apertis is in need of servicing.

2.3 STATUS AND BACKUP FLAGS

The status flags will contain information required by the boot loader to select an appropriate Mini Userspace and for Mini Userspace to select the appropriate subvolume for Full Userspace. It's also used for tracking state at boot time to determine when a failure occurred and trigger a rollback. Listing 1: Flag structure shows the format of the data in the flag partition.

```
struct flag_entry {
    uint32_t mini_version;
    unsigned in_use:1;
    unsigned preferred:1;
    unsigned failed:1;
    unsigned starting:1;
    unsigned running:1;
    unsigned updating:1;
    unsigned factory:1;
    <error correcting codes>
};
```

Listing 1: Flag structure

There's no strict requirement that the flags be stored on the eMMC device, and they could potentially be stored in NOR flash instead – provided both U-Boot and the linux kernel have appropriate drivers to read and write from the NOR device.

It would be possible to store all this information as GPT flags, but there are several

5 http://www.denx.de/wiki/U-Boot/WebHome
6 http://www.denx.de/wiki/view/DULG/UBootScripts
7 http://www.denx.de/wiki/view/DULG/UBootCmdGroupFlash

reasons not to do so:

- GPT flag bits are all “reserved” in the standard and should not be used for arbitrary data.
- Since GPT flags are stored in the partition table, writing to them is a write operation on the table itself. It would be better to minimize writes to the partition table.
- The backup flag data doesn't have to exactly track the primary flag data, so sometimes flags can be updated with only a single write – two would be required if GPT flags were used: one for the primary partition table, one for the backup.
- Depending on hardware configuration, it may be beneficial to store flags in NOR flash instead of on the primary storage device at all. (Though this decision should be very carefully considered due to the frequency of writes to the flag data.)

The first two flag_entries refer to the status of the two Mini Userspace images and the second two entries refer to the status of the Full Userspace BTRFS subvolumes. Some of the fields will have slightly different meanings depending on which entries they're in.

In the entries for the Mini Userspace images, the mini_version will indicate the version of the Mini Userspace image. In the entries for Full Userspace subvolumes it will be the required Mini Userspace version to run that image. The system will never attempt to boot a Full and Mini Userspace pair that doesn't have an exactly matching mini_version.

The in_use flag indicates that an image or snapshot contains valid data. This flag might be 0 if the system was interrupted during an upgrade, or when the system is in a factory state. Apertis will never attempt to boot an image or snapshot that is not in_use.

The preferred flag will be set for exactly one Mini Userspace entry and exactly one Full Userspace entry. These will be the images the system boots under normal conditions.

If the failed flag is set on any partition, then that partition has been flagged as broken and Apertis will not attempt to boot it again. This flag can be written by the system at run-time, or by the boot loader if it is booting in response to the firing of a watchdog timer.

The starting flag is set on an entry before the system begins to launch it, and cleared back to 0 on successful completion. The same write operation that clears starting to 0 will set the running bit to 1. If the system restarts due to a watchdog timeout, these bits will be used to determine when the crash occurred.

The updating flag will be set on an image or snapshot when an update process is started. This will help track updates that must be completed in multiple steps as defined in 3.2 Updating to a new release of the platform. Apertis will never attempt to boot an image or snapshot flagged as updating.

When Apertis is reset to factory state, the factory bit is set on all entries (even ones that aren't in_use). On successful launch of a Full Userspace, the factory flag on the running Full Userspace will be cleared, as the system now has data in the general storage area and is no longer in a pristine factory state. In a system recovery situation, Apertis will consider itself to be in factory state only if all four factory bits are set.

2.3.1 FLASH TRANSLATION LAYERS (FTLS)

The underlying storage system may affect some decisions regarding flag data size and location, so a brief explanation of flash translation layers is provided here. A deeply technical article on flash memory design can be found at https://wiki.linaro.org/WorkingGroups/Kernel/Projects/FlashCardSurvey, this document attempts to touch on a few of the simpler points.

- Raw flash devices have poor longevity – they wear out as they're written to.
- Data on raw flash is usually grouped into large blocks called “erase blocks”, and any write operation must start by first clearing an entire erase block.

To combat these problems, a translation layer will mediate requests from the operating system to the flash storage. Writes will be transformed into combinations of read, erase, and write operations to mask the large size of the erase blocks. Writes to the same virtual address on the device will result in access to different physical block addresses in an effort to distribute the damage caused by writing to the device -- a process called “wear leveling”.

How the FTL performs these operations can have an impact on device reliability and performance. Reliability when a power failure occurs during a write is especially important, as on some devices it may result in the loss of all data in an entire erase block. Due to wear leveling, two blocks of data with very different addresses may actually be stored in the same erase block.

Good wear leveling algorithms will extend the life of the underlying flash significantly, but with the negative side effect that when the first block fails all the rest will fail very soon after. Some flash products include health monitoring logic to alert the operating system that the safe number of write cycles has been exceeded and the device will fail in the near future.

Some of the decisions in the following chapter may need to be tailored to the specific flash device and FTL that Apertis is deployed on in order to minimize the possibility that both flag blocks can be corrupted at once (either by power failure or by a device approaching the end of its service life) and to minimize the possibility that flag data and user data are stored in the same erase blocks.

2.3.2 FLAG ROBUSTNESS

Since the boot flags are critical to successfully starting the system, care is taken to protect their contents and restore them if they've been damaged.

There are two possible failure modes for flag data:

- The flags can not be read at all due to a device error.
- The flags can be read but are wrong.

To protect against device errors, a secondary copy of the flags will be stored. The capabilities of the storage device may have some influence on the ideal locations and sizes for the primary and backup flags, but generally the layout will be similar to Illustration C: One Possible On-Device Layout of Flag Data.

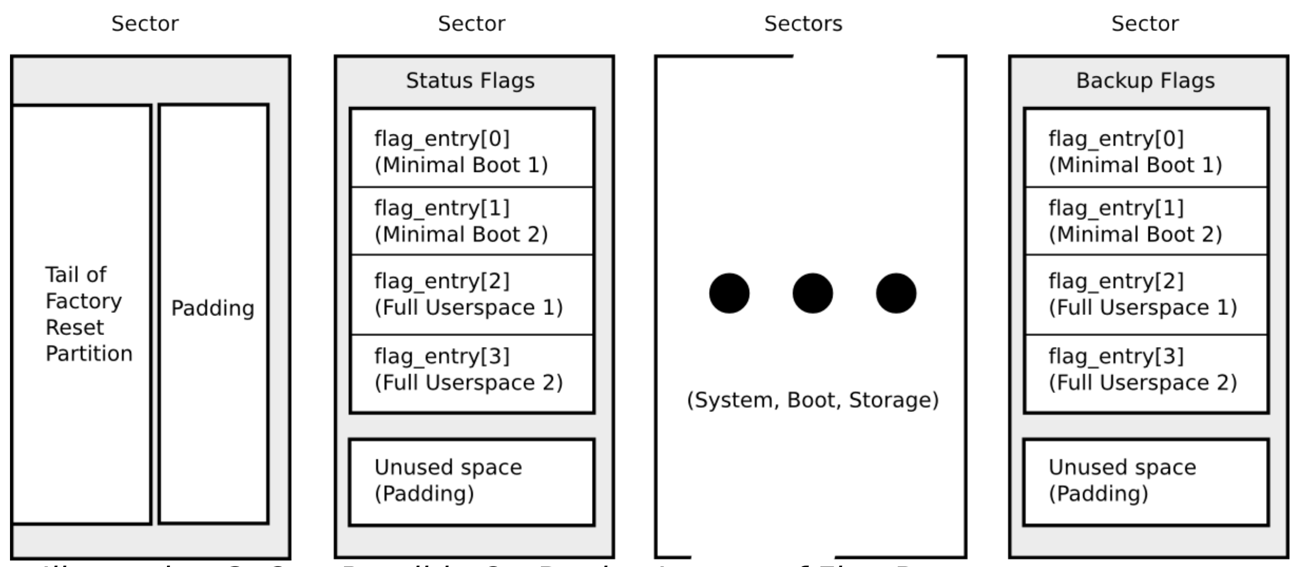

Sector
Sector
Sectors
Sector
Tail of Factory Reset Partition
Padding
Status Flags
flag_entry[0] (Minimal Boot 1)
flag_entry[1] (Minimal Boot 2)
flag_entry[2] (Full Userspace 1)
flag_entry[3] (Full Userspace 2)
Unused space (Padding)
(System, Boot, Storage)
Backup Flags
flag_entry[0] (Minimal Boot 1)
flag_entry[1] (Minimal Boot 2)
flag_entry[2] (Full Userspace 1)
flag_entry[3] (Full Userspace 2)
Unused space (Padding)

Illustration C: One Possible On-Device Layout of Flag Data

The backup copy will be updated far less frequently than the primary flags – it won't track starting or running states. It is hoped that this slight decoupling of primary and backup will reduce the possibility of both copies failing at once.

To detect readable but incorrect flag data, a series of sanity checks will be performed. Flag states that don't make any sense – such as both Full Userspace subvolumes flagged as preferred simultaneously – will be used to detect problems with the flag data.

The structures will also be protected by a Hamming(8,4)[8] code. This code was chosen to meet the following criteria:

- It can be quickly encoded or decoded using small look-up tables – the combined storage for both tables is 528 bytes.
- It works on small symbols (4 bit when encoding, 8 when decoding), so the implementation can be exhaustively verified for correctness with an automated test.
- It is a SECDED code – Single Error Correcting, Dual Error Detecting. For every 8-bit input symbol, any single bit error can be corrected and any

8 http://en.wikipedia.org/wiki/Hamming_code#Hamming.287.2C4.29_code_with_an_additional_parity_bit

error consisting of 2 incorrect bits can be detected.

Incorrect bits in the flags aren't expected to be a common occurrence, so a simple and fast code such as this should provide a slight increase in robustness without any performance loss or risk of incorrect decoding due to a faulty implementation.

In the event that the primary flags are unreadable or inconsistent, the backup flags will be used to decide which partition to boot, and the contents of the primary flags will be overwritten.

If the failure was due to unreadable data, overwriting the damaged flag data should result in the sector being remapped, and the error will disappear. If the primary flag block remains broken then the storage device is likely at the end of its life, and the system health monitoring software (See the robustness document for details) will take appropriate action and inform the user of a need for servicing before further storage failures begin to manifest.

2.4 RECOVERY TOOL

The first partition on Apertis will be a bootable system recovery partition capable of overwriting the flags, minimal boot and system partitions with a factory fresh state and re-formatting the user storage area. This partition will never be updated, and is intended to return the device to the same state it was originally shipped in.

The recovery tool will check for the presence of an image file in the recovery partition – if no such image is present, it will prompt for an image to be provided on a USB stick[9]. This allows OEMs to optionally omit the recovery image for large space savings without completely losing the ability to do a factory restore.

In the event that no Mini Userspace can be booted, the factory fallback image will be automatically attempted, potentially saving the user from having to bring their vehicle in for servicing.

2.5 MINI USERSPACE

The Mini Userspace boot partition would include the kernel and enough infrastructure to display a welcome screen - which may include the display of a rear view camera if the vehicle is in reverse gear and is equipped with appropriate hardware.

Once the welcome screen is displayed, Mini Userspace would use status information from the state partition to determine whether a rollback needs to be performed. After the optional rollback procedure, It would update the status flags accordingly, and start the Full Userspace image.

2.6 FULL USERSPACE

Full Userspace implements the majority of the operating system, and all of the

9 The installation process will be similar to the one in 3.2 Updating to a new release of the platform, and follow the same security procedures to prevent installation of unverified software.

interactive components. As with Mini Userspace, it will be mounted read-only. Any run-time storage requirements will be met by tmpfs for small volatile items such as sockets or temp files or persistent storage in the general storage area for logfiles and user installed applications.

Transitions between Mini and Full Userspace will be visibly smooth and flicker free. If the welcome screen is animated, it may be extended until Full Userspace finishes starting.

2.7 SWAP SPACE

A swap partition is conspicuously absent from (Illustration A: Storage layout) as there are many reasons to avoid the use of swap space entirely:

- Swap is slow, and depending on it to make a ram starved system usable will result in a poor user experience.
- Flash storage devices have a limited number of write cycles, so actively using swap will reduce the lifetime of the hardware.
- The Linux kernel may use swap space when not strictly necessary – having swap available on a device that doesn't need it can reduce performance. Some usage patterns that provoke this behavior – playing large quantities of media from the storage device – are likely to be common on Apertis.
- If a buggy program uses too much memory a system with swap may “thrash”[10], spending all of its time performing input/output operations, delaying the inevitable killing of the offending process while a swapless system may recover much more quickly.
- Swap space permanently reduces the amount of storage space available to the user.

However, one must be pragmatic. Some devices may be equipped with less than optimal amounts of memory in order to hit lower price points, and swap may be required to make these configurations usable.

BTRFS does not currently support swap files, so if swap space is required for a system to operate, a permanent swap partition must be created.

One alternative to conventional swapping is zRAM[11], which reserves a large chunk of RAM to be used as compressed swap space. This approach is beneficial because it doesn't degrade the flash storage device rapidly, and because it's significantly faster than swapping on flash. However, it is still slower than not having swap at all, and most importantly is still considered experimental code.

10 http://en.wikipedia.org/wiki/Thrashing_(computer_science)
11 http://en.wikipedia.org/wiki/ZRam

3 UPDATE MANAGEMENT

Update management for applications and for the base system is an important aspect of the lifecycle of the Apertis, especially since it is a goal to keep the system very current and fresh, getting updates every so often. Applications will be updated through a mechanism described in the Application Design – for convenience, a subset of that information is provided in 6 Application Management .

System updates can be categorized in two kinds:

- Updates for components of the current version of the platform, with bug and security fixes, and eventually minor functionality improvements;
- Platform updates, that change the version of the platform; the main goal of such updates is to bring in new functionality.

These two kinds of updates are very different in nature, and require different handling. Small updates for bugs and especially security fixes can be time-critical, but also relatively small and hopefully relatively quick to download and install, while an update to a new version of the platform is less time-critical (assuming there is still security support for the old version), but will also take more time. For both cases the system should regularly check for the availability of updates and indicate to the user when these are available.

The foundations of the features discussed in this chapter already exist: the Debian package manager dpkg and the APT library and its tools are able to validate repositories, check sums for packages, download packages from several different kinds of sources, such as HTTP servers or file systems.

However, the user interface and mechanisms required to implement the fail-safe upgrade process need to be implemented. Collabora will provide the backend of the update manager for Apertis, most likely implemented as a D-Bus service which will implement the policies set by the applications. This daemon will be responsible for managing the snapshots, checking for updates, downloading in the background and so on.

It should be mentioned that there are tools such as debdelta[12] designed to minimize the data sent for package upgrades, however as they're not enabled by default in Debian/Ubuntu, they haven't seen significant real world testing. To determine if such an optimization is even worthwhile, the tools should be tested on the specific packages in the Apertis image. Currently Collabora suspects that there aren't particularly large gains to be made here, and a potential for pushing corrupt packages exists.

3.1 UPDATES FOR COMPONENTS OF THE PLATFORM

Component updates can automatically be downloaded in the background, constrained by any network policies, like only allowing downloads while on wi-fi, or unlimited 3G. When the download is complete, the user will be notified the updates are ready to install, and be able to initiate it.

12 http://debdelta.debian.net/

As shown in Illustration D: Update Procedure, the update manager will perform the following steps:

- Create a new snapshot of the current system-platform
- Mount the snapshot, and chroot into it
- Perform the upgrade inside the chroot
- Update the flags in the flag partition to boot into the new system volume

From the perspective of the update manager, a kernel upgrade is the same as the installation of any other package. Pre-installation and Post-installation scripts in the kernel package will take care of installing the new boot files in the appropriate partition. Since the upgrade process could be interrupted after the new kernel is installed, the flags will not be finalized until the upgrade process finishes. This prevents an interrupted upgrade from ever resulting in a system attempting to boot a kernel that is incompatible with its userspace.

The system will begin using the new components the next time it's started. While it's technically possible to drop back to Mini Userspace briefly and restart only Full Userspace, this would take a few seconds and the process couldn't be completely hidden from the user.

If for any reason the update process fails to complete, the update will be blacklisted to avoid re-attempting it. Another update won't be automatically attempted until a newer update is available.

It's possible that an update will be successfully installed, yet fail to boot, resulting in a rollback. In the event of a rollback the update manager must be notified that the new update has not been correctly booted, and blacklist the update, so it is not attempted again.

This policy will prevent a device from getting caught in a loop of rollbacks and failed updates at the expense of running an old version of the system until a newer update is pushed. The update management framework will provide a method for clearing a blacklist if it is chosen to expose such functionality in a GUI.

The most convenient storage location for the blacklist is the user storage area, since it can be written at run-time. As a side effect of storing the blacklist here, it will automatically be cleared if the system is reset to a factory state or if the user storage area is wiped.

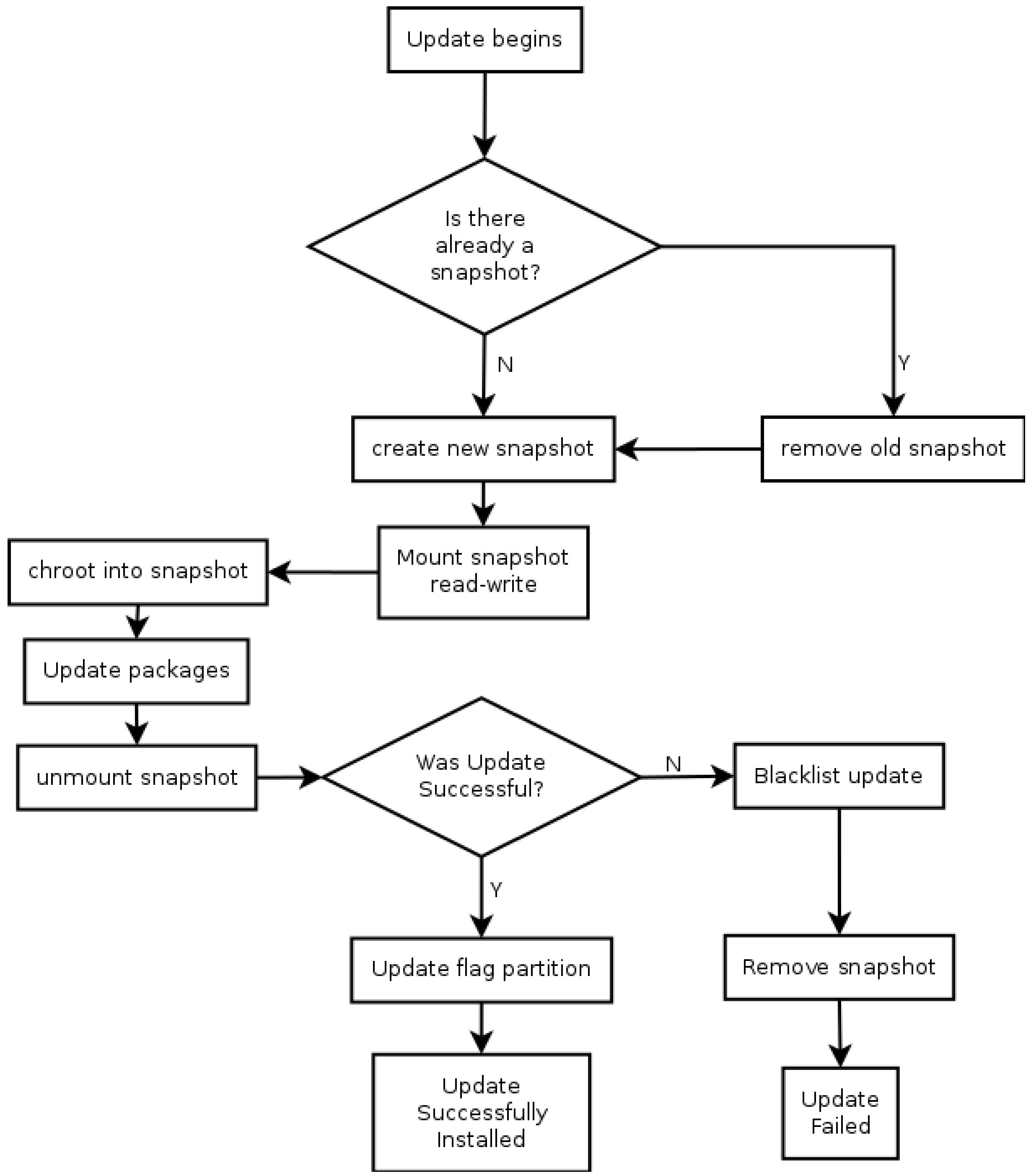


Illustration D: Update Procedure

3.2 UPDATING TO A NEW RELEASE OF THE PLATFORM

For system-wide upgrades Collabora recommends using a full image replacement instead of doing package upgrades. This does not mean the same mechanism used for updating components individually could not be used, but full image replacement enables to take a clean slate approach when doing bigger upgrades, upgrades from older versions straight to the newest one, or simply sidestepping complex upgrades that might prove challenging to do with the regular mechanism.

Shipping compressed filesystem images may result in a larger than strictly necessary amount of network traffic – theoretically empty space in the images may not actually be easily compressible. For this reason tarballs will be the preferred distribution system for full system updates.

A single archive contaning the full system may be too large to fit in available storage, so for convenience an update should be split into multiple individually signed pieces. These pieces will be verified and installed sequentially, potentially over multiple days depending on the driver's vehicle usage. Details of the archive creation process are covered in Chapter 8: File Archiving and Platform Updates.

The system will know about new versions by periodically downloading a well-known URI to obtain a file hosted on the packages repository that indicates what versions exist. Apertis will, while checking for component upgrades, download this file and check if any new information is available. When a new version is made available the file will indicate that fact, and the Apertis shall then present the user with the choice of upgrading to the new platform release.

The file used by Ubuntu can be seen here: http://changelogs.ubuntu.com/meta-release. Notice how all Ubuntu releases are listed, every entry also has URIs pointing to the *Release* file, used by APT, the release notes, which are shown to the user before upgrading, and an upgrade script.

Using Ubuntu releases as an example, to upgrade from Warty Warthog (4.10) to Oneiric Ocelot (11.10) would require upgrading to each full release in order (ie: first to the latest version of Warty Warthog, then from there to Hoary Hedgehog, Breezy Badger, etc etc etc) and would take a considerable amount of time and network traffic.

This situation could occur on Apertis if a user didn't have a network connection for a long time and then decided to update, or if they performed a factory reset on an old device and subsequently wanted to bring it up to date.

In addition to providing time and traffic savings in these circumstances, there is also a reduction in QA load – it removes the requirement to test automated updates from very old versions of the system every time a new version is released.

While possible to implement, a system-wide upgrade done through the APT packaging system adds a lot of complexity without providing any significant features to Apertis. When a new version is detected by the update manager, the user will be presented with the choice of initiating the upgrade.

When the user accepts, the system will start downloading the first piece of the new system image to the data partition. The download will respect connectivity policy restrictions, such as only downloading while on wifi or unlimited 3G connections. The update manager may provide the user with a way of overriding connectivity policy restrictions.

Upon completion, the following steps will be followed to perform the upgrade:

1. The system subvolume that isn't currently in use will be flagged as “updating” and then be deleted.

2. A new empty system subvolume will be created in its place.
3. The fetched piece file will be verified by digital signature to be legitimate.
4. New system files from the piece file will be unpacked inside the new volume.
5. The piece file will be deleted, and if more pieces exist, the next will be fetched.
6. Steps 3 through 5 will be repeated until no additional pieces remain.
7. If a new Mini Userspace image is to be installed, the updating flag will be set on one of the minimal boot partitions, and its contents replaced.
8. If everything was successful then anything marked as updating will now be made preferred.

While performing the steps above the update manager will keep a state file that it updates before each step, so that it can recover from a power loss and restart from the last known good state.

After completing the full installation, the next time Apertis is restarted it will be running the new software. After booting into the new release successfully any leftover files used for installation will be removed by the update manager.

While no mention has yet been made of updates touching the general storage area, it is possible that after a system update some components installed there will require additional updates as well.

After the system is started with the new version software, the application update system will notice any new versions available that required the system update. Add-ons such as themes and skins could be updated through the application store mechanisms at this point.

If built-in applications have their own interfaces for downloading data into the user storage area (perhaps a mapping application has an in-app interface for downloading area specific maps), then they should begin any required updates of their own data on the first launch of the new version.

It is hoped that these two mechanisms will provide enough flexibility that the system update framework never has to upgrade anything stored in the general storage area.

3.3 SERVER-SIDE SUPPORT FOR SYSTEM UPDATES

To provide updates a server infrastructure is required. An HTTP or FTP server can be used to serve the metadata and package files of what is often called an APT repository. As part of the Apertis project Collabora will set up and maintain an internal repository to be used for development and testing. Full details of this infrastructure are provided in the Build & Integration design, and only a cursory overview will be provided here.

As discussed in the security design, there is a chain of trust that needs to be validated so that only authentic and valid packages reach the device. The root of that chain is the APT repository, which provides a *Release* file, which contains the hashes for all *Packages* files and is signed with a PGP key whose public counterpart is shipped along with the device.

For system-wide upgrades a simple HTTPS service will be required to provide the meta-release file and the system images. For greater security the images can be accompanied by a detached signature (a signature that goes in a separate file) using the same PGP key as the one used to sign packages, so that it can be verified by the update manager before starting the upgrade.

3.4 SYSTEM UPDATES FROM EXTERNAL MEDIA

It is possible that using the networks that are available to Apertis may not be a convenient method for installing updates. It may not have a fast connection available at all, there might not be enough space in the internal file systems for the download or it may be in one of many cars that must be upgraded. Using external media for such upgrades is an option.

This functionality would work in a similar way to the ones described above. User interface will need to be specified, but following is a description of how it could work.

The user would download an image from a web site and put it in the root of an external drive. After the drive is mounted, the update management daemon would look for files with a specific name pattern in the root of the drive. If an appropriate file is found, the update management daemon would mount the file using a loopback[13] block device and verify that it's a signed package repository or collection of piece files that comprise a system image which has a newer version than the currently installed system. If that verification passes, the user would get a notification saying that updates are available from the drive.

Upon telling the system that the upgrade should be started the user would be asked not to remove the drive, and a similar upgrade procedure as described in the previous sections would be carried out (with the notable exception that the files will be used directly from the loopback device instead of dowloaded to internal storage), with an announcement at the end that it is safe to remove the drive. If the user ignores the previous warning and removes the drive while the upgrade is still running the upgrade will be stopped.

The procedure would be continued or restarted the next time the user inserts an external drive carrying an update. Since the online and offline upgrade methods use the same piece files, it would be possible to continue upgrades using a different method than they are started with. Whether this is truly useful or not is debatable, but it is a side-effect of providing a stateful upgrade system that can continue an upgrade over a reboot.

3.4.1 SECURITY CONCERNS FOR EXTERNAL MEDIA UPGRADES

Updates from external media present a security problem not present for directly downloaded updates. Simply verifying the signature of a file before

13 http://en.wikipedia.org/wiki/Loop_device

decompressing it is an incomplete solution. A user with sufficient skill and resources could create a malicious USB mass storage device that presents different data during the first and second read of a single file – passing the signature test, then presenting a different image for decompression.

Care will be taken during the installation to only read the contents of the mass storage device once, performing verification and decompression in a single pass. This could potentially result in invalid files being written to an inactive system volume, but the boot flags will never be set to allow launching the invalid subvolume.

3.5 UPDATE PROCEDURE TEMPORARY STORAGE REQUIREMENTS

As previously stated, at run-time, system storage areas will be kept read-only. This is good for security and reliability reasons, but presents problems to the update tools, which require temporary writable storage.

Traditionally, package management tools keep a local cache of packages on the root file-system which may be used to perform rollbacks. Our snapshot mechanism removes the need for long term storage of this cache, but the package dependency resolver still needs all the packages for an update to be locally available before installation begins. Collabora proposes using temporary space in the general storage area for this cache, and deleting it once the update is finished.

For component upgrades, the update manager knows in advance how much space is required to store the packages. If insufficient space is available before the download begins, or if the user fills the available space before the package download finishes, the user will be asked to provide external storage or free up some space.

For full system upgrades, it's possible that there won't be enough available space in the data partition to download even a single piece of a full system image. In that case, the user will be made aware that an update is available, and asked to perform the update from external storage.

Since full system upgrades will be downloadable in pieces, it would be possible to permanently reserve enough storage space on the system to hold a single one of these pieces so that a user is never forced to update from external media.

3.6 THE POST-UPDATE RESTART

In order to begin running newly installed system components, a restart is required. To keep the update process seamless to the user, the system will not be automatically restarted. The user will begin using the upgraded software the next time the system is turned on.

Even a trivial system update will require a restart before it becomes active, but since Apertis will be powered down from time to time (when the car is parked, for example), the system won't spend much time running outdated software.

4 SYSTEM SNAPSHOTS AND ROLLBACK

When an upgrade is started a snapshot of the system partition will be performed. The update manager will then chroot into the newly created snapshot to perform the actual upgrade in the background. Once that finishes successfully the flag partition will be updated for both the new Full Userspace and the new Mini Userspace to be preferred.

If the upgrade operation fails during package installation the process can be easily reverted by simply removing the new snapshot, and the update can be blacklisted, to avoid insisting on a broken upgrade. If the update fails while installing a new Mini Userspace, the flag partition will not be updated and the snapshot space can be reclaimed.

Regardless of technology used, empty space will need to be kept free on the storage to perform snapshots. For BTRFS this means keeping empty space in the file-system, and for LVM this means keeping free extents in the volume group. No more than 50% of the space reserved for the main system should ever be in use, and a new version of the system also needs to comply with that rule. When the system is being upgraded, a snapshot will be created and it is possible that no blocks will be shared by the new system and the old one, by the end of the upgrade.

```
root@goiabinha:/test/lvm# vgdisplay
  --- Volume group ---
  VG Name               vg0
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  4
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               1
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               952.00 MiB
  PE Size               4.00 MiB
  Total PE              238
  Alloc PE / Size       200 / 800.00 MiB
  Free  PE / Size       38 / 152.00 MiB
  VG UUID               gQuXN3-BN9l-IJNl-1wd3-iFWM-lbWl-F3PLIP

root@goiabinha:/test/lvm# lvcreate -s /dev/vg0/lvm -n new_snapshot -L40M
  Logical volume "new_snapshot" created
root@goiabinha:/test/lvm# vgdisplay | egrep '(Free  PE|Alloc PE)'
  Alloc PE / Size       210 / 840.00 MiB
  Free  PE / Size       28 / 112.00 MiB
```

Listing 2: How space is managed when using LVM

2 shows how space is managed in the volume group. The *Free PE* line tells us how many extents the logical volume still has to be used. When creating the snapshot, LVM needs to be told how much space it needs to reserve for its copy-on-write needs (40MB in the example); every time something changes in a snapshot that space is used. Each snapshot should have as much space reserved for it as the main file-system, since there's a potential that many blocks will be different. The same goes for new volumes created for system-wide updates: they should be sized with the size of the whole image in mind.

This covers essentially all there is to it, functionality wise; however, the hard question on this matter is what technologies should be used. While BTRFS shows up as a natural choice given its feature set, there are some risks that need to be considered. The following sections raise the main issues at hand, and investigate the possible solutions.

4.1 THE CASE FOR AND AGAINST BTRFS

BTRFS has several great features that can be used to make a platform like Apertis more robust and simpler to use. Its built-in support for having copy-on-write, first-class citizen snapshots makes it very simple and effective to provide recovery from bad system upgrades. However, one of the key attributes one would expect in such a critical piece of the system infrastructure is still missing: maturity. While various distributions have made plans to switch to BTRFS as a default file-system, none have done so yet[14].

Fedora, known as a bleeding edge distribution which is always an early adopter for technologies, intended to use BTRFS as the default file-system for their F16 release (Released November 2011), but gave up since it wasn't ready yet for prime time. Fedora F18 release may enable it by default[15].

Past experience has shown that widespread use in a popular distribution is an important step for a file-system and its tools to become properly production-ready. Stability fixes are still going into mainline quite often. The 3.3 merge window included many such fixes. Collabora believes that will be the case until at least 3.5. The current version of the kernel of choice for the project is the 3.2 version provided by Linaro. Should BTRFS be adopted it would be very important to have a much newer kernel, otherwise a significant number of backports will be required.

In early 2012 both Oracle[16] and SUSE[17] have begun shipping linux distributions with BTRFS support.

For now, the project will assume the use of BTRFS for specification purposes. Any

14 MeeGo has been pushing BTRFS as their default for a while, but no products have been released using it; the Nokia N9, the most popular and last MeeGo product to be released uses ext4, for instance.

15 https://fedorahosted.org/fesco/ticket/704

16 https://emeapressoffice.oracle.com/Press-Releases/Oracle-Announces-Production-Release-of-Unbreakable-Enterprise-Kernel-Release-2-for-Oracle-Linux-29a4.aspx

17 http://www.suse.com/releasenotes/x86_64/SUSE-SLES/11-SP2/#fate-306585

development effort will be delayed as much as possible to accommodate a later decision on what to do.

4.2 A FALLBACK IF BTRFS IS NOT READY FOR PRIME TIME

Providing a subset of the features BTRFS has can be done by using a mature file-system along with the LVM technology. A large amount of the functionality described in this document – with some minor limitations imposed - can be implemented with LVM instead of BTRFS, so some detail on LVM is provided here. However, what is written in the Applications Design makes heavy use of BTRFS functionality that can't be adequately emulated with LVM.

If BTRFS is found unsuitable, major changes will be required in the Applications Document, and some of the functionality described in this document will need to be revised. This information is presented as a starting point should significant difficulties arise in the use of BTRFS, but is not a complete solution.

LVM snapshots work very much like BTRFS ones. No data is copied at snapshot creation, so creation is pretty much instantaneous. The snapshot does not use significantly more space than BTRFS's snapshots, the main difference in this regard is that the space to be used by snapshots needs to be set aside explicitly, while BTRFS uses whatever space exists in the file system.

LVM snapshots are also backed by the same blocks as the original volume until they are changed in one of them, like in BTRFS, so they are also very efficient and quick to create.

There is one challenge when implementing this fail-safe upgrade with LVM, though. LVM snapshots, unlike BTRFS ones are not first-class citizens. The snapshots depend on their *origin,* which is the original logical volume. When time comes for the previous file system to be removed the process is not very straight-forward, and needs to be handled with caution: the snapshot needs to be merged back into the original file system, which will then receive all changes that were performed to the snapshot.

While this operation is not atomic and cannot happen while any of the two logical volumes is open, it does run in the background after the next LVM activation, and is able to survive power losses and system failures. Unfortunately, even though the merge takes place in the background, a new snapshot can not be created until the merge operation finished. When time has come to remove the old system volume, **lvconvert** would be called like this:

```
root@goiaba:~# lvconvert --merge /dev/vg0/system2
  Can't merge over open origin volume
  Merging of snapshot system2 will start next activation.
```

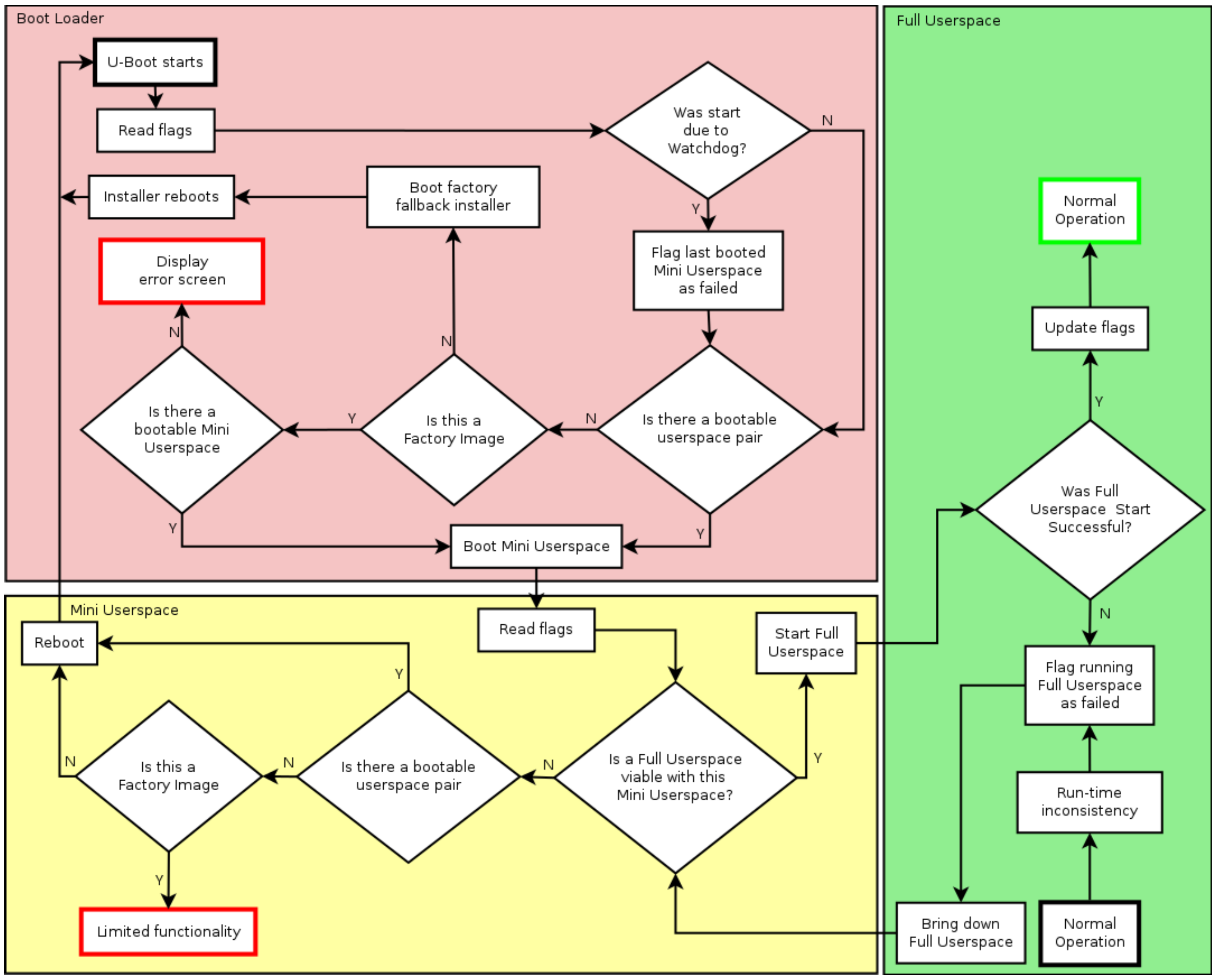
Listing 3: Merging back a snapshot as a means of removing the old system volume

In summary, to perform an upgrade, if a snapshot is present the system must restart once to begin reclaiming the snapshot. After the reboot the system will be functional (possibly with reduced storage performance) but unable to begin the upgrade until the merge completes – a procedure that will take a variable amount of time based on the size of the previous upgrade.

This additional complexity may influence the decision about whether and for how long the old system volume should be kept. This should be better detailed on the technical specification.

4.3 THE ROLLBACK PROCESS

Illustration E: Rollback Procedure


Boot Loader
U-Boot starts
Read flags
Was start due to Watchdog?
N
Y
Installer reboots
Boot factory fallback installer
Flag last booted Mini Userspace as failed
Display error screen
N
N
Is there a bootable Mini Userspace
Y
Is this a Factory Image
N
Is there a bootable userspace pair
Y
Y
Boot Mini Userspace
Y
Full Userspace
Normal Operation
Update flags
Y
Was Full Userspace Start Successful?
N
Flag running Full Userspace as failed
Run-time inconsistency
Bring down Full Userspace
Normal Operation
Mini Userspace
Reboot
Read flags
Start Full Userspace
Y
N
Is this a Factory Image
N
Is there a bootable userspace pair
N
Is a Full Userspace viable with this Mini Userspace?
Y
Y
Limited functionality

As shown in Illustration E: Rollback Procedure, a rollback can be triggered in a couple of ways. In the case of a kernel failure, the watchdog timer will force a reboot. Depending on device configuration, this timer may be implemented by external logic or directly on the CPU. The boot loader will query the appropriate mechanism (CPU status registers, GPIO lines) to determine whether to boot normally or provoke a rollback.

If a running system determines something is functioning poorly (a “run-time inconsistency” in the illustration), it may provoke a rollback. This might be triggered if the file-system is corrupt, if built-in applications fail to launch properly, or core pieces of infrastructure malfunction. More details are provided in the Robustness Design.

Inconsistencies in the status flags (see 2.3.2 Flag Robustness) may result in a return to factory state if they can't be resolved, though this (along with numerous flag updates) has been left out of the diagram to keep it simple.

The system will first try to boot the preferred Mini Userspace/Full Userspace pair. If either of these are flagged as failed, the system will evaluate the possible combinations of Full Userspace snapshots and Mini Userspaces to determine if a functional system can be achieved.

In the event that no Full Userspace snapshots can be made to function, the system will revert to a factory clean state, replacing the current system images and flag partition as well as deleting all user data.

If the hardware is failing, reverting to a factory clean state will not be sufficient to restore proper operation. If the watchdog hardware fires while the system is in a factory clean state, the factory partition will be marked as failed. While recovery is no longer an option, it may still be possible to provide very limited functionality or inform the user that the system is in need of servicing.

In this situation, the factory Mini Userspace image will be started in an attempt to provide very limited functionality (perhaps enabling the use of a rear-view camera) while displaying a notice that the driver should bring the vehicle in for service.

If the factory Mini Userspace is no longer functional at all, the boot loader will attempt to display a static image informing the user that the equipment is in need of servicing.

5 USER AND USER DATA MANAGEMENT

As described in the MultiUser Design, Apertis should be multi-user, to accommodate the needs of more than one driver. Collabora envisions the multi-user support as essentially using the existing concepts and features of the several open source components that are being adopted, with some fine tuning.

Except for a few general concerns, Collabora does not see this document as the appropriate place to describe how specific user data such as photos, music, and so on should be handled. That should be handled instead by the Media and Indexing Design and the Applications Design. That said, the following are important points to keep in mind.

All user data will be kept in the general storage area (see Illustration A: Storage layout). As discussed in the Security Design, this setup enables simpler separation of concerns, and a simpler implementation of user data removal.

Since the general storage area is the only writable storage space, it must also house data (such as tracker's database) that isn't tied to a specific application. This data may be affected by version changes, so configuration should be kept in a separate subvolume in the general storage area that will be snapshot and rolled back along with the system.

5.1 EXPOSING SNAPSHOT CAPABILITIES TO APPLICATIONS

It has been considered exposing file-system snapshot functionality to applications through the middleware. The application installation framework requires this functionality, and it must exist to allow the installer described in the Applications Design to function. This chapter deals with the details of offering similar functionality to application developers for performing atomic operations.

To make finely grained snapshots (rather than snapshots of the entire user storage area), each application would need to be able to create new subvolumes to store data in. Under LVM, applications would have to know in advance how large these volumes could become – most likely either underestimating and being unable to complete their operations, or overestimating and wasting storage space. Large amounts of unallocated space would need to be kept in a pool for applications to allocate from.

Under BTRFS no advanced allocation or free space pools are required, but considerable infrastructure and policy would need to be implemented to allow the system to safely reclaim application snapshots – either when an application is uninstalled, or if a poorly written application doesn't properly manage its own snapshots.

The snapshot and rollback system for applications would also need to deal with the snapshots an application may have created on its own, in order to preserve the application's exact state in case a rollback is desired.

Given the complexity of implementing such a system, a fully general purpose interface to the snapshot mechanism is likely very difficult to implement. However, by narrowly defining the applications abilities, it may be possible to present useful functionality without creating difficult problems.

By exposing two capabilities from the SDK: “open checkpoint” and “close chekpoint”, applications can be given an interface to safely perform long operations on their data sets without risk of corruption from a system shutdown.

The open function would create a BTRFS snapshot of the application's subvolume. At this point the application could begin making high risk changes to its files. Once the changes are complete, a call to the close function would simply delete the snapshot created by the open function.

Should the system be brought down for any reason in the middle of this operation, on next startup the boot procedure would notice the presence of a checkpoint. It would delete the partially changed subvolume and rename the checkpoint subvolume in its place.

A third capability - “abort atomic operation” is actually difficult to implement, but might be important enough to develop anyway. The difficulty stems from the fact that the changes are being made to a mounted subvolume – in order to revert the changes the subvolume must be unmounted, deleted, and replaced with the snapshot.

The application must be closed before the operating system will allow unmounting the subvolume it's running from, so an abort would necessitate stopping the running application and any running subcomponents (such as an an agent), at which point the subvolume could easily be replaced by the checkpointed version.

6 APPLICATION MANAGEMENT

There are two types of applications: Store and Built-in. Built-in applications are part of the system image, and will be updated by the system package management tools.

Application management on Apertis has requirements that the main package management system does not:

- It is unreasonable to expect a system restart after an application update.
- Each application must be tracked independently for rollbacks. System updates only track one “stream” of rollbacks, where the application update framework must track many.
- The Apertis application rollback paradigm is novel, and no existing tools closely match its requirements.

The full details of handling applications are explained in the Applications Design where these problems and other related issues are addressed. Relevant details of application storage and updating is presented here for completeness.

At the time of this writing, some details of application management are not yet finalized, and this document attempts to avoid duplicating information that may become incorrect through further revisions of the Applications Design.

6.1 SYSTEM APPLICATION STORAGE

Since multiple users may use the same application, to save space the applications will be installed per device, not per account. Store applications will be stored in the read-write storage partition in an application storage directory. User data will be stored within the application storage directory as well so it can be snapshot and rolled back with the associated application.

Built-in applications will be stored in a directory with the same structure, but in the read-only system storage area (under /usr/Applications). Their user settings and data will be stored in the read-write storage partition.

The details of the directory structures are in the Applications Design.

6.2 APPLICATION STORAGE

An application may require storage space for personal settings, license information, caches, and any manner of long term private storage. These files should generally not be easily accessible to the user as directly modifying them could have detrimental effects on the application.

Application storage requirements can be divided into broad groups:

- An area for application extensions to system functionality. This will be managed by the application manager and not directly by applications themselves.
- User specific application data – for settings and any other per-user files.

In the event of an application rollback, files in this area stored in the same BTRFS subvolume as the application itself, and are automatically rolled back with their associated application.

- Application specific application data – for data that is rolled back with an application but isn't tied to a user account – such as voice samples or map data. This data is stored in the same way as user specific application data.
- Cache – easily recreated data. To save space, this won't be stored for rollback purposes, and will be cleared on a rollback in case applications change their cache data formats between versions.
- Storage for files in standard formats that aren't tied to specific applications, as explained in the Multiuser Design, this storage will be shared between all users. This data will be exempt from the rollback system.

All of these will be inside the general storage subvolume, and more detail can be found in the Applications Design.

6.3 UPDATE AND ROLLBACK PROCEDURE

The intent of the application rollback system is to allow a user to rollback any application, along with all of its settings and data, to the previous version. The intent is to guarantee a user the ability to roll back the most recent update to an application should they prefer the old version for any reason. Extra versions may be saved when the system is updated to ensure a system rollback won't render applications inoperable.

Built-in applications are updated by the system update tools, and no mechanism will be provided for the user to roll them back – they are implicitly rolled back as part of a system rollback.

When either type of application is updated, their associated storage and setting subvolumes will be snapshot. Only one snapshot will be kept, so any older snapshots will be destroyed. Cache will be cleared on an update or rollback.

Should the user request an application rollback, the current application, configuration and data will be destroyed and the old version's restored – it will be as if the new version of the software had never been installed.

The implementation of this feature may involve taking a snapshot of the application files at the time of an update, or it may involve keeping the application bundle around and only taking snapshots of user settings and data. The most efficient method will depend on the size of the application and the amount of change between versions.

When an application is uninstalled, all copies saved for rollback purposes are deleted. OEMs will have control over whether application settings are saved when applications are uninstalled – the default will be to keep user settings, but delete system settings.

6.4 EFFECT OF SYSTEM ROLLBACK ON INSTALLED APPLICATIONS

If the system is rolled back, applications that were dependent on infrastructure not available in the older system may fail to function properly – they may behave unpredictably, or if libraries are no longer present they may not even be able to execute.

The system “API level” is a version number used by the application management system to prevent applications that don't work with the system version from being installed or launched.

Collabora will provide an API for querying the API level of the system so an application can try to work across multiple versions of the system, but as entire libraries may not be present, this is only a partial solution.

To attempt to reduce the chance that a system rollback causes applications to fail to launch, the system will attempt to roll back all applications whose API level is newer than the currently running system, if they have available rollback copies with a compatible API level.

Storing an extra application rollback point when a system update occurs should make this possible much of the time, but when the rollback is successful the user may lose settings and data as part of the rollback process.

Any applications that were Installed more recently than the last system update may still be broken by a system rollback.

Since they are stored in the system subvolume, built-in applications will always work after a system roll-back, so the essential system software will always be functional.

6.5 EFFECT OF ROLLBACK ON APPLICATION DEVELOPMENT

On a conventional Linux system, some infrastructure (such as the gsettings configuration system) can be configured to store information pertaining to multiple applications in a single database. However, storing configuration for multiple applications in the same file confounds the rollback system, making it much more difficult to roll application settings back independently of eachother. Storing settings for applications in multiple files in the individual application directories will ease the rollback process.

Since the rollback system requires a slight change from the way Linux applications usually store data, the SDK can provide generic API for storing settings and files. This would hide some of the complexity and help developers store settings and data in the right place.

7 SETTINGS AND DATA MIGRATION

When an Apertis user purchases a new vehicle, they may wish to transfer their existing settings, data, and applications to the new system. Implementing this functionality has some potential pitfalls:

- If the old vehicle has the latest version of the Apertis system, but the new vehicle doesn't, it is possible that the settings for built-in applications won't be compatible.
- To avoid accidentally making software piracy easy, care must be taken to confirm that the new system is allowed to run the applications to be migrated.
- If applications are used during the migration process, transferred data may be in an inconsistent state.
- If the new Apertis has a newer version of the system software, the older Apertis's applications may be obsolete and need updates to function. This situation may be worse if the user has intentionally rolled back an application, and is now forced to use a newer version they dislike.

Some of these problems are trivially resolved – by making the migration tool modal and disallowing normal device access during the process, data consistency is ensured. This also has the benefit of not requiring any special volume snapshot management functionality to be available to the migration tool.

Version conflicts and software licensing issues are more difficult to handle, and the solution may not be completely satisfying to all users. Requiring the new system to be updated to the latest platform version before beginning the migration process will prevent the built-in applications from being presented settings newer than the software.

In order to protect against software piracy, it is recommended that the migration tool scan the installed applications on the old Apertis and attempt to download the latest versions from the application store to install on the new Apertis. The application store will refuse to generate application bundles that shouldn't be installed on the new Apertis. Only the settings and the data – not the actual applications – will be copied from the old device.

The biggest drawback of this method is that a user will be forced to upgrade to the latest software versions when transferring settings, even if they'd previously chosen to rollback an application.

8 FILE ARCHIVING AND PLATFORM UPDATES

With the exception of occasional improvements in compression formats, file archiving is generally considered a solved problem. However, some of the requirements of Apertis make the available solutions less than ideal for distributing replacement system images.

Apertis needs a file archiving system for platform updates. The important requirements are:

- Large system distributions can be split into smaller "piece files"
- these small piece files can be installed sequentially in multiple sessions, possibly across multiple reboots
- Only one piece file is required at a time.

If these requirements are met, Apertis will be able to perform a full platform update with minimal temporary storage requirements and without requiring the vehicle to be kept running for a lengthy single session update procedure.

Many open source archival tools (such as pax, tar, or cpio) encapsulate files by adding a header to the front of each file to store file meta-data (such as creation time and permissions). These files are then concatenated into a single larger file.

Compression is usually performed on the entire archive, but could instead be performed on a file by file basis before the files are placed in the archive. Illustration F: File Encapsulation shows how files are stored and compressed by tools such as tar and gzip.

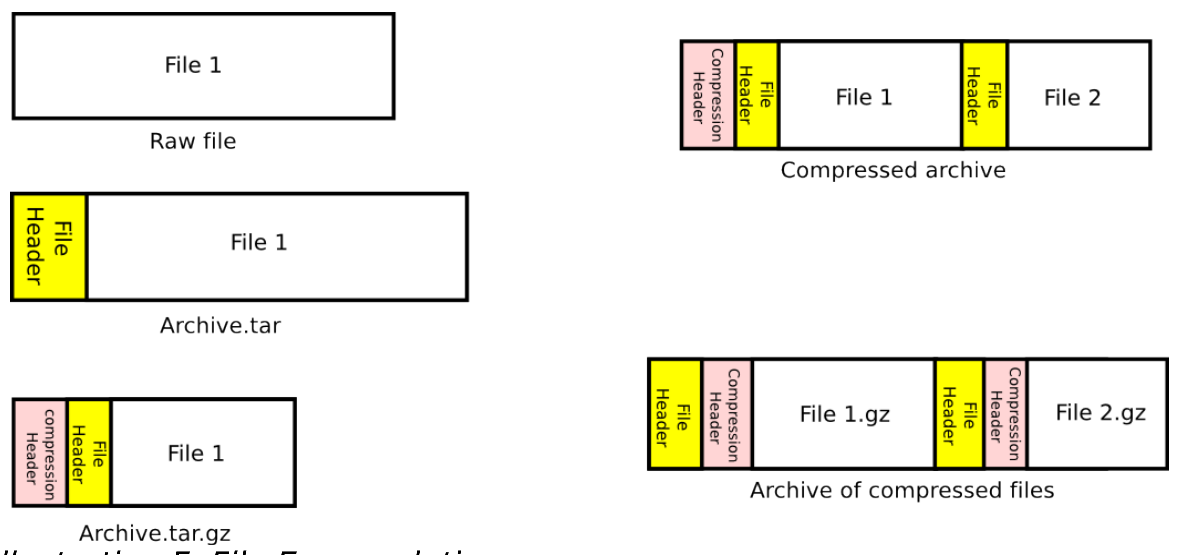


Illustration F: File Encapsulation

The conventional way to create a multi-part archive (Illustration G: Conventional Archive Splitting) is to naively split the archive file into equal sized pieces, paying no attention to file boundaries.

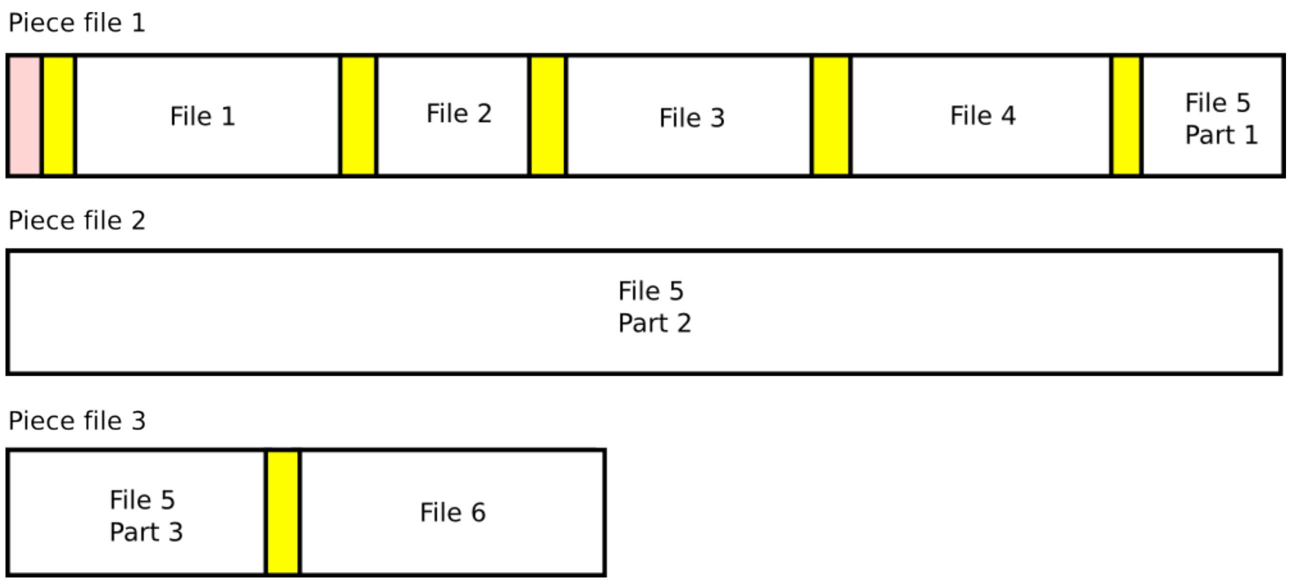


Illustration G: Conventional Archive Splitting

In the usual case this is an excellent way of splitting archives – it results in no extra overhead, all the split files except the last are the same size, and the size of the split files can be arbitrarily chosen.

However, it has a major drawback from the perspective of Apertis system updates: Only the first file has header information at the start. The practical implication of this is that it would not be possible to perform an installation across multiple system restarts, since the restoration tool would have no way to understand the second file without reading the first.

A very simple way to solve this problem with existing tools, is shown in Illustration H: Splitting on File Boundaries. The files are compressed individually before being added to the archive, and a new archive is started when adding a new file would exceed the intended maximum file size.

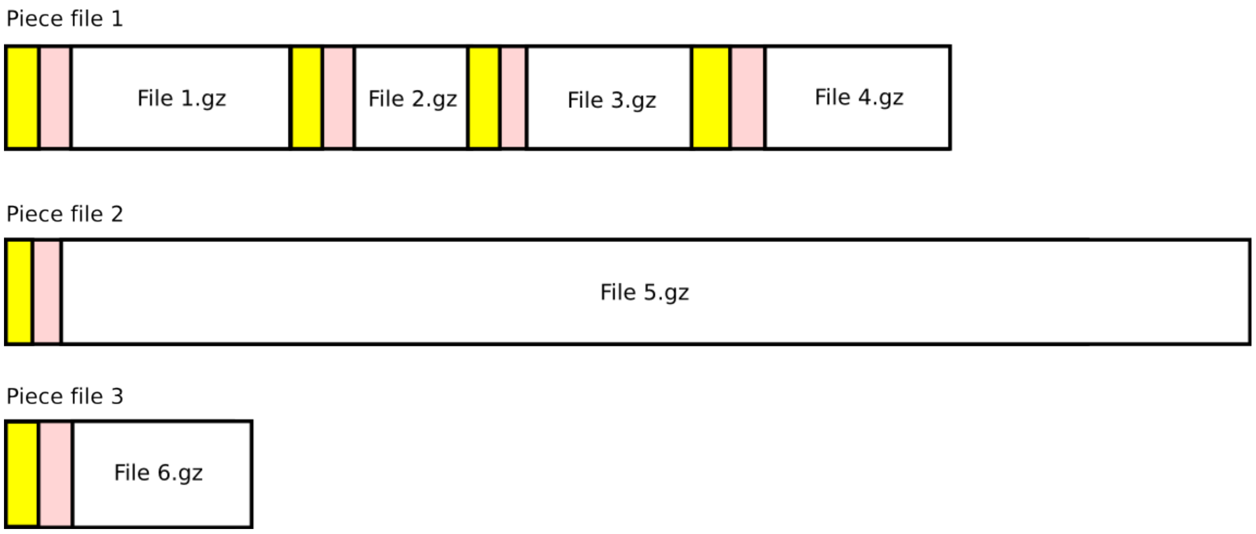


Illustration H: Splitting on File Boundaries

This approach is also not without problems:

- Since individual files can't be split, the compressed size of the largest file to be archived dictates the minimum size for the split files.
- Compressing each file individually reduces the effectiveness of the compression algorithms as well as increasing the space consumed by compression headers.
- If each file is compressed individually, extra space is required to decompress them after they're extracted from the archive in a compressed state.

The reason for compressing the files before adding them to the archive is that it is not known in advance how well a file will compress. There are several alternatives (such as performing calculations based on uncompressed sizes, performing multiple passes, or making assumptions about the compression efficiency) but each solves the problem by introducing other problems.

Another alternative method would be splitting large files before adding them to the archive. Since a single input file might result in multiple files to be archived, a simple pipeline of existing tools wouldn't be possible. For example, the “split” tool is incapable of writing split files to stdout – this is because stdout and stdin are “streams” and can't convey file boundaries in any standard way.

Files needing re-assembly would need to be flagged as such - perhaps by cataloging them in a file or by adding a suffix to their filenames. After extraction a recombining step would need take place. Theoretically, this recombining step requires extra space to be reserved on the target filesystem, but by carefully ordering the files during the archival process it can be ensured that the last pieces to decompress don't contain split files. This will prevent having to recombine files when the filesystem is near full.

While no changes to existing tools would be required, new software would need to be written to perform the pre and post processing steps. Another undesirable feature is that the files need to be compressed before being added to the archive (otherwise it is impossible to choose a reasonable boundary for the split). Since the large files would be split after compression, only the first part would contain a header – if a file spanned many pieces, the entire file would need to be recombined before it could be decompressed.

Collabora has extending an existing archival tool and written a special splitter tool to allow the splitting method shown in Illustration I: Implemented Splitting Method.

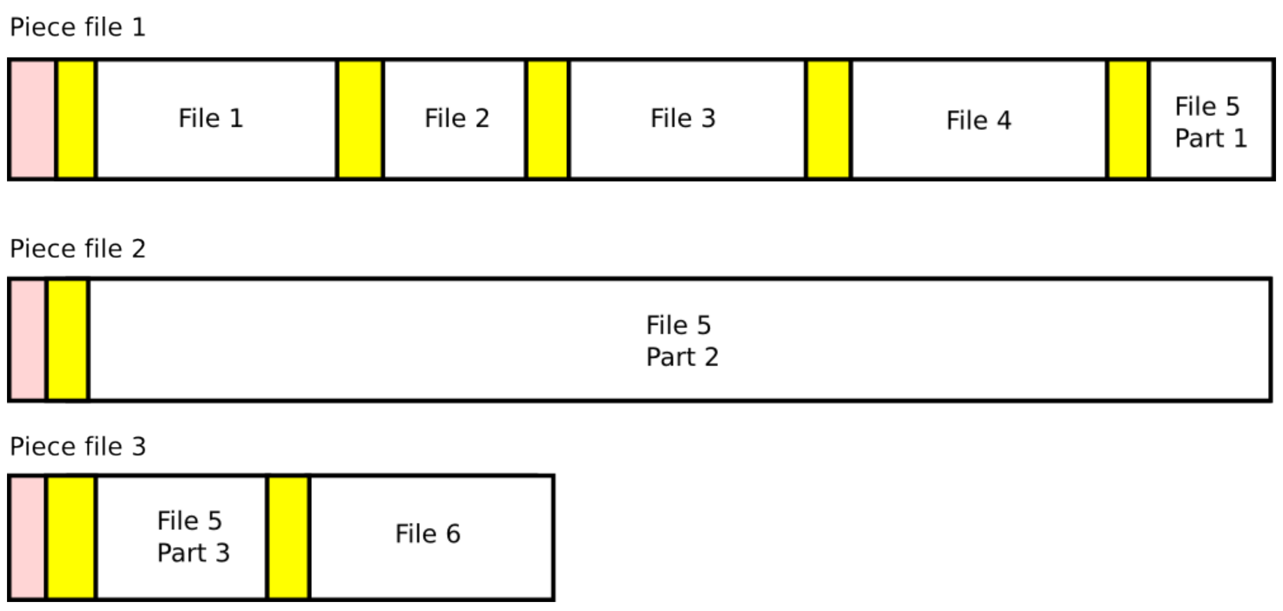


Illustration I: Implemented Splitting Method

Tar, a widely distributed archival tool, is used to create a single file archive from the contents of a system image. A new tool has been written to split this archive and compress the split piece files.

Tar archives precede each contained file with a header indicating their length. The files must end with a “footer” to tell tar no more data is available. This means that breaking a tar file at an arbitrary location will not result in a proper tar file – tar will return an error code at the end of extraction.

To avoid this, when a file is broken by the splitting tool, it zero-fills to the expected file size, then includes the appropriate footer. The zero data will compress to almost nothing, so the resulting file will be within a few bytes of the intended size yet still be a valid tar archive.

A new “type” value[18] of 'A' is present for the first header of a piece that begins with a continued file. This indicates that the file_size field in the header is actually an offset to begin inserting new data at.

18 See http://en.wikipedia.org/wiki/Tar_(computing)#UStar_format for details on the tar header

A special modified version of GNU tar has been created that understands this slightly altered format. The special version of tar is only used for system update procedures, an unaltered version of tar is present on the system for any regular use that might occur.

Using this method allows the process of extracting files to be restarted after an interruption, and allows selection of a reasonable maximum file size independent of the size of any of the files to be stored. It also allows files to be inserted into the archive before compression, which results in simpler extraction.

The extraction procedure is simply passing each piece to this modified tar in order. Files that are broken across archive boundaries are filled as shown in Illustration J: Split File Extraction.

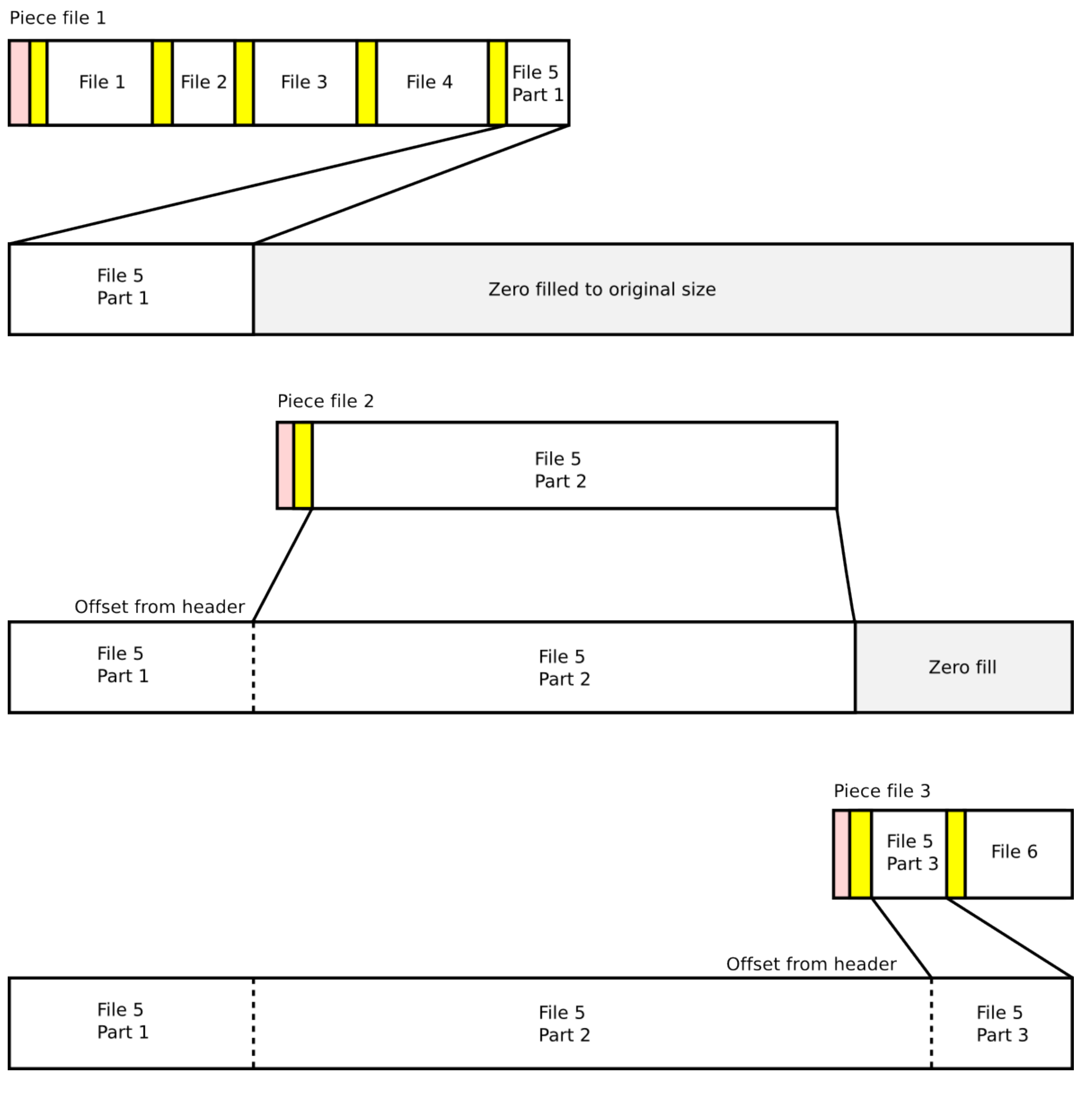


Illustration J: Split File Extraction

If extraction is interrupted for any reason, the process can be resumed by starting at the beginning of the piece file that was interrupted. This allows whole platform system updates (as described in 3.2, Updating to a new release of the platform) to be resumable in the event that Apertis is shut down before the update can be completed.

9 RELATED DOCUMENTS

The following documents are referenced by, and useful in the interpretation of, the information in this design:

- The Applications Design
- The Supported API Design
- The Build and Integration Design
- The Media Indexing Design
- The Multi-User Design
- The Security Design

As well, the following URLs contain pertinent information:

U-Boot boot loader:

- http://www.denx.de/wiki/U-Boot/WebHome
- http://www.denx.de/wiki/view/DULG/UBootScripts
- http://www.denx.de/wiki/view/DULG/UBootCmdGroupFlash

Debdelta package manager/service:

- http://debdelta.debian.net/

BTRFS development and support:

- http://www.suse.com/releasenotes/x86_64/SUSE-SLES/11-SP2/#fate-306585
- https://emeapressoffice.oracle.com/Press-Releases/Oracle-Announces-Production-Release-of-Unbreakable-Enterprise-Kernel-Release-2-for-Oracle-Linux-29a4.aspx
- https://fedorahosted.org/fesco/ticket/704