Making a more resilient file system

Colophon

Release	2
Build date	07/08/2026
Build version	2a4426fec072

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Document version history

Release	Date	Description
1	22 Dec 2021	Initial release
2	22 Jun 2024	Added section on pSLC for CMx
3	7 Aug 2026	All round improvements bringing document up to date with current state of the art

Scope of document

This document applies to the following Raspberry Pi products:

Single Board Computers / SBCs

Pi Zero			Pi Zero 2		Pi 1				Pi 2	Pi 3			Pi 4	Pi 5
-	W	H	W	WH	A	B	A+	B+	B	A+	B	B+	B	-
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Compute Modules

CM0	CM1	CM3	CM3+	CM4	CM4S	CM5
✓	✓	✓	✓	✓	✓	✓

Keyboard Computers

Pi 400	Pi 500	Pi 500+
✓	✓	

Introduction

Raspberry Pi devices are frequently used as data storage and monitoring devices, often in places where sudden power downs may occur. As with any computing device, power dropouts can cause storage corruption. This white paper provides some options on how to prevent data corruption under these and other circumstances by selecting appropriate file systems and setups to ensure data integrity.

This white paper assumes that the Raspberry Pi is running Raspberry Pi OS and is fully up to date with the latest firmware and kernels.

What is data corruption and why does it occur?

Data corruption refers to unintended changes in computer data that occur during writing, reading, storage, transmission, or processing. In this document we are referring only to storage, rather than transmission or processing. Corruption can occur when a writing process is interrupted before it completes, in a way that prevents the write from being completed, for example if power is lost.

It is worthwhile at this point to give a quick introduction to how the Linux OS (and, by extension, Raspberry Pi OS), writes data to storage. Linux usually uses 'write caches' to store data that is to be written to storage. These cache (temporarily store) the data in random access memory (RAM) until a certain predefined limit is reached, at which point all of the outstanding writes to the storage medium are made in one transaction. These predefined limits can be time and/or size related. For example, data may be cached and only written to storage every five seconds, or only written out when a certain amount of data has accumulated. These schemes are used to improve performance: writing a large chunk of data in one go is faster than writing lots of small chunks of data.

However, if power is lost between data being stored in the cache and it being written out, that data is lost.

Other possible issues arise further down the writing process, during the physical writing of data to the storage medium. Once a piece of hardware (for example, the Secure Digital (SD) card interface) is told to write data, it still takes a finite time for that data to be physically stored. Again, if power failure happens during that extremely brief period, it's possible for the data being written to become corrupted.

When shutting down a computer system, including a Raspberry Pi, best practice is to use the `shutdown` option. This will ensure that all cached data is written out, and that the hardware has had time to actually write the data to the storage medium.

The SD cards used by the majority of Raspberry Pi devices are great as cheap hard drive replacements, but they are susceptible to failure over time, depending on how they are being used. The flash memory used in SD cards has a limited write cycle lifetime, and as cards approach that limit they can become unreliable. Most SD cards use a procedure called 'wear levelling' to ensure they last as long as possible, but in the end they can still fail. This can be from months to years, depending on how much data has been written to or, more importantly, erased from the card. This lifetime can vary dramatically between cards. SD card failure is usually indicated by random file corruptions as parts of the SD card become unusable.

There are other ways for data to become corrupted, including, but not limited to, defective storage media, bugs in the storage-writing software (drivers), or bugs in the applications themselves.

For the purposes of this white paper, any process by which data loss can occur is defined as a corruption event.

What can cause a write operation?

Most applications do some sort of writing to storage, for example saving configuration information, database updates, and the like. Some of these files may even be temporary, i.e. only used while the program is running, and do not need to be maintained over a power cycle; however, they still result in writes to the storage medium.

Even if your application does not actually write any data, Linux will constantly be making writes in the background, mostly writing logging information.

Hardware solutions

Although not totally within the remit of this white paper, it is worth mentioning that preventing unexpected power downs is a commonly used and well-understood mitigation against data loss. Devices such as uninterruptible power supplies (UPSs) ensure that the power supply remains stable. If power is lost, the UPS switches to battery power and notifies the computer system that power loss is imminent, allowing shutdown to proceed gracefully before the backup power supply runs out.

Because SD cards have a limited lifetime, it may be useful to have a replacement regime that ensures SD cards are replaced before they have a chance to reach end of life.

Robust file systems

There are various ways that a Raspberry Pi device can be hardened against corruption events. These vary in their ability to prevent corruption, with each action reducing the chance of it occurring.

Reducing writes

Simply reducing the amount of writing that your applications and the Linux OS do can have a beneficial effect. If you are doing lots of logging, then the chances of writes happening during a corruption event are increased. Decreasing logging in your application is down to the end user, but logging in Linux can also be reduced. This is especially relevant if you are using flash-based storage (e.g. eMMC, SD cards) due to their limited write life cycle.

Journaling

Most writable file systems used on Raspberry Pi devices, including ext4, use a technique called 'journaling' to protect against corruption. Before making a change to the file system's structure (creating a file, deleting a file, extending one, and so on), the file system first writes a record of the intended change to a dedicated area of the disk called the 'journal'. If a corruption event happens part way through the real update, the file system can replay the journal on the next mount to complete or roll back the interrupted change, bringing the file system's structure back into a consistent state.

Note

Journaling protects the **structural consistency** of the file system: the integrity of its metadata, directory entries, and free-space accounting. It does not guarantee the **content** of an individual file that was being written to at the time of the corruption event. That file can still end up truncated or otherwise incomplete even on a fully journaled file system — see the 'Application-level data integrity' section in this document to learn how to protect against this.

Ext4 supports three journaling modes, set using the `data=` mount option:

- `data=ordered` (the default) — metadata is journaled, and file data is guaranteed to be written to storage before the metadata that references it. This is a good balance of safety and performance, and is the recommended mode for most uses.
- `data=writeback` — metadata is journaled, but file data can be written to storage in any order relative to it. This is faster, but after a corruption event a file's metadata may reference data that hasn't actually been written yet, which can expose old, unrelated data inside what should be a newly written file.
- `data=journal` — both metadata and file data are journaled. This is the safest mode, but also the slowest, since it roughly doubles the amount of data written to storage for every change.

Warning

Avoid `data=writeback` unless you fully understand the risk. It trades a real, if fairly narrow, safety guarantee for a modest performance gain, and can expose the previous contents of freed disk blocks inside a file after a corruption event.

errors=remount-ro

By default, ext4 uses `errors=remount-ro`. If the kernel detects corruption or an inconsistency in a mounted file system, this option immediately remounts it as read-only rather than allowing further writes that could make the damage worse. This is a sensible default and should not normally be disabled: it will not fix a corruption problem, but it will stop it from getting worse and prevent applications from acting on a file system that is already known to be damaged.

Write barriers

Ext4 uses 'write barriers' (`barrier=1`, the default) to enforce the order in which data is physically committed to storage, even though the storage hardware and its drivers may otherwise cache and reorder writes for performance. Without barriers, the ordering guarantees the journal depends on to recover cleanly after a corruption event no longer hold.

Various online guides suggest disabling write barriers (`barrier=0`) as a way of "tuning" SD card performance.

Warning

Do not disable write barriers. Doing so directly undermines the crash-consistency guarantees this white paper sets out to achieve, while offering only a small performance gain compared to other mitigations described here, such as increasing the commit time (see below).

Detecting corruption: `metadata_csum`

Journaling protects against inconsistency caused by an interrupted write, but it does not detect corruption introduced elsewhere, for example a bit error caused by flash wear or a failing SD card. The `metadata_csum` feature, enabled by default when a modern `mkfs.ext4` creates a file system, adds checksums to ext4's metadata structures so that this kind of corruption can be detected — and in some cases identified and repaired by `e2fsck` — rather than silently causing incorrect behaviour.

Changing commit times

Linux uses two related, but distinct, mechanisms that affect how quickly written data reaches storage. These are easily conflated:

- The generic kernel 'dirty page writeback' timer (`vm.dirty_expire_centisecs`, 5 seconds by default) controls how long modified data can sit in the page cache, for any file system, before the kernel starts writing it out.
- Ext4's own 'journal commit interval' (`commit=`, 5 seconds by default) controls how often ext4 forces a journal commit, checkpointing recently written data and metadata to storage together.

Increasing the `commit=` value improves performance by batching up more writes into each commit, but it also increases the amount of data that can be lost if there is a corruption event before the next commit. Reducing it means there is less chance of a corruption event leading to data loss, although, as covered above, it doesn't prevent it completely.

To change the commit time for the main ext4 file system on Raspberry Pi OS, you need to edit the `/etc/fstab` file, which defines how file systems are mounted on startup.

Code

```
$sudo nano /etc/fstab
```

Add the following to the ext4 entry for the root file system:

Code

```
commit=<commit time in seconds>
```

So, `fstab` may look something like this, where the commit time has been set to thirty seconds. The commit time will default to five seconds if not specifically set.

Code

```
proc            /proc           proc    defaults          0       0
PARTUUID=50a7611a-01  /boot           vfat    defaults          0       2
PARTUUID=50a7611a-02  /               ext4    defaults,noatime,commit=30  0       1
```

Note

Thirty seconds is a reasonable starting point on devices where flash wear is the bigger concern: it cuts write frequency well below the five-second default, roughly matching the kernel's own dirty-page writeback timer without leaving an excessively long window of vulnerability. Pick a shorter commit time if minimising data loss after a corruption event matters more to you than flash wear, bearing in mind the wear cost discussed in the 'Reducing writes' section above and the pSLC section below.

Temporary file systems

If an application requires temporary file storage, i.e. data that is only used while the application is running and not required to be retained across shutdown/reboot, then a good option to prevent physical writes to storage is to use a temporary file system, 'tmpfs'. Because these file systems are RAM-based (actually, in virtual memory), data written to tmpfs does not by itself affect flash lifetimes or become damaged during a corruption event.

Note

This assumes swap is not in use. tmpfs is backed by virtual memory rather than being a genuine RAM disk, so its pages are just as swappable as any other anonymous memory: under memory pressure, the kernel can write tmpfs contents out to a swap device on physical storage. If swap is enabled, data 'written' to a tmpfs mount can still end up on flash storage, undermining both the wear and corruption event safety benefits described here. Either disable swap on devices where this matters, or ensure enough RAM is available that tmpfs pages are not swapped out under normal operation.

Creating a tmpfs location for temporary application data requires editing the `/etc/fstab` file, which controls all the file systems under Raspberry Pi OS. The following example replaces the storage-based `/tmp` location with a temporary file system:

Code

```
tmpfs /tmp tmpfs defaults,noatime 0 0
```

Logging to RAM

Raspberry Pi OS uses systemd-journald for system logging, so mounting a separate tmpfs over the whole `/var/log` directory is a coarser, older technique that predates journald's own support for RAM-only storage. Rather than hiding everything under `/var/log` behind a separate mount, journald can be told directly to keep its journal in memory only, via a drop-in configuration file:

Code

```
mkdir -p /etc/systemd/journald.conf.d
cat > /etc/systemd/journald.conf.d/volatile.conf <<'EOF'
[Journal]
Storage=volatile
RuntimeMaxUse=16M
EOF
```

This has the same practical effect as the tmpfs mount above - RAM-only, size-capped, no writes to physical storage - without affecting anything else that might live under `/var/log`.

Note

`Storage=volatile` (like the tmpfs mount) discards all logs on reboot, including anything logged immediately before a corruption event - exactly the information you are most likely to need to diagnose what happened. If that matters, keep the journal persistent instead, but tune it to bound its size and write frequency so it doesn't undermine the rest of this document's advice:

Code

```
mkdir -p /etc/systemd/journald.conf.d
cat > /etc/systemd/journald.conf.d/persistent.conf <<'EOF'
[Journal]
Storage=persistent

# Reduce space
Compress=yes

# Lower disk budget
SystemMaxUse=512M
SystemMaxFileSize=20M

# No time window retention
MaxRetentionSec=0
MaxFileSec=0

# Endurance profile
RuntimeMaxUse=128M
SyncIntervalSec=2m
RateLimitInterval=30s
RateLimitBurst=2000
EOF
```

There is also a third-party script that helps set up logging to RAM, which can be found on GitHub. This has the additional feature of dumping the RAM-based logs to disk at a predefined interval.

Root file system type

The root file system ('rootfs') is the file system on the disk partition on which the root directory is located, and it is the file system on which all the other file systems are mounted as the system is booted up. On a Raspberry Pi it is at `/` and is a read/write (RW) ext4 file system. The directory containing assets required for device boot, such as kernels, DTBs, VideoCore firmware, etc., is mounted RW on `/boot/firmware` and is a FAT file system (see 'Protecting the boot partition' below).

Read-only ext4 root file system

Mounting the rootfs as read only (RO) prevents any write accesses to it, making it much more robust to corruption events. However, unless other actions are taken, this means nothing can write to the file system at all, so saving data of any sort from your application to the rootfs is disabled. If you need to store data from your application but want a read-only rootfs, a simple technique may be to add a USB memory stick or similar that is just for storing user data. A better technique is to include a RW file system and bind mount into it from the RO rootfs on the directories you wish to write to.

Note

If you are using a swap file when using a read-only file system, you will need to move the swap file to a read/write partition.

Native read-only file systems: SquashFS and EROFS

The approach above takes an ext4 file system, designed to be written to, and mounts it as read-only. An alternative, and increasingly common, approach on embedded Linux systems is to build the rootfs as a *native* read-only image in the first place, using a file system such as SquashFS or EROFS (Enhanced Read-Only File System).

Both SquashFS and EROFS are:

- **Immutable by construction** — there is no write support at all, so there's no risk of an application or a stray process corrupting the image, and no journal is needed, since nothing on the file system ever changes while it's mounted.
- **Compressed** — images are typically much smaller than an equivalent ext4 file system, which also reduces the amount of data that needs to be read from storage.
- **Built offline** — the image is created as a single file (for example with `mksquashfs` or `mkfs.erofs`) from a source directory tree, then flashed or written to a partition, rather than being created and populated live on the device.

EROFS is the more modern of the two, designed to improve on some of SquashFS's performance limitations: it offers better random-access performance, and is used as the read-only system partition file system on Android and on rpi-image-gen AB images. Both are supported in the Linux kernels that Raspberry Pi ships.

This is also standard practice when building embedded images: rather than retrofitting read-only behaviour onto an ext4 image built for a desktop-style OS, the whole rootfs is built from the start as an immutable, compressed image.

Overlay file system

An overlay file system (overlayfs) combines two file systems, an 'upper' file system and a 'lower' file system. When a name exists in both file systems, the object in the upper file system is visible while the object in the lower file system is either hidden or, in the case of directories, merged with the upper object.

Raspberry Pi provides an option in `raspi-config` to enable an overlayfs on top of the standard Raspberry Pi OS ext4 rootfs. This makes the ext4 rootfs read only as the lower file system, and creates a RAM-based (tmpfs) upper file system for any writes. This gives a very similar result to the read-only ext4 file system described above, with all user changes being lost on reboot.

You can enable an overlayfs using either the command line `raspi-config` or the desktop Raspberry Pi Configuration application on the Preferences menu.

Note

The standard pattern for an overlayfs-based embedded system is to use a native read-only image (SquashFS or EROFS) as the lower file system, with a tmpfs or a small dedicated writable partition as the upper file system. This differs from the `raspi-config` overlayfs option, which uses an ext4 lower file system marked read-only rather than a genuinely immutable image, and so doesn't get the compression, size, and robustness benefits of a native read-only image format.

There are also other implementations of overlayfs that can synchronise required changes from the upper to the lower file system at a predetermined schedule. For example, you might copy the contents of a user's home directory from upper to lower every twelve hours. This limits the write process to a very short space of time, meaning corruption is much less likely, but does mean that if power is lost before the synchronisation, any data generated since the last one is lost.

Adding a persistent read-write data partition

None of the read-only approaches above remove the need to store data that must survive a reboot: configuration state, logged readings, calibration data, and so on. The common pattern is to leave the rootfs untouched, no matter how it's implemented (read-only ext4, overlayfs, or a native read-only image), and add a small, separate read-write partition purely for this data, either on the same SD card or eMMC, or on external storage such as a USB drive.

Because this partition is now the **only** thing on the device being written to, it's worth applying all of the mount-option tuning described earlier in this document specifically to it: an appropriate journaling mode (`data=ordered`), the default write barriers, `metadata_csum`, and a commit time chosen to balance data loss against flash wear (see 'Journaling' and 'Changing commit times' above). A small, dedicated data partition like this is also easier to reason about than a whole read/write rootfs, since it's the only place that storage writes — and so corruption events — can occur.

Note

This is different from the tmpfs-based upper file system used by overlayfs, described above. An overlayfs upper file system is RAM-based, and its contents are lost on reboot; the writable data partition described here is on physical storage, and is intended specifically to persist.

Flash-friendly file systems: F2FS

Ext4 was designed with spinning hard disks in mind, and although it works fine on flash storage, it has no particular awareness of flash's underlying characteristics. F2FS (Flash-Friendly File System) is a Linux file system designed specifically for NAND flash-based storage such as SD cards and eMMC, and is available on Raspberry Pi OS.

F2FS is aware of the underlying properties of flash storage — in particular, that flash is erased in large blocks before it can be rewritten. It uses a log-structured design, writing data sequentially to new locations rather than modifying data in place, which reduces the amount of read-erase-write cycling ('write amplification') that a general-purpose file system such as ext4 can cause on the underlying storage. F2FS also implements its own wear-levelling and background garbage collection.

Warning

F2FS's real-world track record is mostly on Android phones, where shutdown and power state are tightly controlled by the OS and hardware, rather than on devices that are expected to lose power unexpectedly. Corruption following an unclean shutdown is more commonly reported with F2FS than with a well-tuned ext4, which is a far more battle-tested file system for exactly this scenario. Since this white paper is specifically about surviving unexpected power loss, ext4 — with the journaling and mount option tuning described above — is the safer default for RW. Only consider F2FS if you have validated its behaviour under power-loss testing representative of your own deployment.

Protecting the boot partition

The sections above focus on the rootfs, but the boot file system is worth considering separately: it is a FAT (vfat) partition, and by default remains read/write even when the rootfs is made read-only, an overlayfs is used, or corruption events elsewhere are otherwise mitigated against.

This matters because FAT has no journal: if a corruption event interrupts a write to `/boot`, there is no recovery mechanism to bring it back into a consistent state. The risk is not purely theoretical either — firmware and configuration updates (for example to `config.txt`, or kernel and device tree updates) all write to `/boot`, making it a plausible real-world location for corruption. Unlike a corrupted user data file, a corrupted `/boot` partition can prevent the device from booting at all.

Practical ways to reduce the risk to `/boot` include:

- Mounting `/boot` as read-only, only remounting it as read-write for the short periods when OS, firmware, or configuration updates actually need to be applied.

- Minimising how often such updates are applied, particularly on devices already deployed in the field.
- On eMMC devices specifically, it may be possible to utilise 'power-on write protection' on this area. Protecting the partition table this way provides additional robustness. The rpi-image-gen AB layout does this for eMMC configurations.

pSLC on Raspberry Pi Compute Modules

Pseudo single-level cell (pSLC) is a technology that can be enabled on compatible eMMC storage devices to trade capacity for write endurance. This permanently reconfigures a region of the device so that each cell stores only a single bit, rather than the two (MLC) or three (TLC) bits it would normally hold.

Note

Raspberry Pi Compute Modules have historically featured multi-level cell (MLC) eMMC, but newer models are transitioning to triple-level cell (TLC) eMMC to help secure supply and pricing. See the *Transitioning from MLC to TLC flash memory on Raspberry Pi Compute Modules* white paper for a full comparison of MLC and TLC endurance, error correction, and firmware behaviour, along with a detailed treatment of pSLC itself, including how to determine pSLC/enhanced user area status from software.

pSLC is implemented on eMMC as an enhanced user area (also known as enhanced storage) — a feature that is defined by the JEDEC eMMC standard, but whose physical implementation is left up to the vendor.

Warning

Configuring a region as pSLC does not, by itself, guarantee improved reliability. Some vendors write pSLC cells using true single-level voltage states, meaningfully increasing endurance and data retention; others implement enhanced user areas in a way that provides little or no benefit over the device's standard operation. Users can contact Raspberry Pi Ltd for advice on whether pSLC is appropriate for a specific application and eMMC part — see the *Transitioning from MLC to TLC flash memory* white paper referenced above.

Implementation details

- Enhanced user area is a concept, whereas pSLC is an implementation.
- pSLC is one way of implementing an enhanced user area.
- At the time of writing, not all eMMC devices used on Raspberry Pi Compute Modules implement the enhanced user area.
- There is no need to configure the entire eMMC user-data area as an enhanced user area; partial enablement is supported.
- Enabling pSLC reduces the usable capacity of the area it's applied to. The fraction lost depends on the eMMC's native cell type — see the *Transitioning from MLC to TLC flash memory* white paper for the specifics of MLC versus TLC.
- Programming a memory region as an enhanced user area is a **one-time operation**, and should be treated as an irreversible decision made during provisioning — before an operating system is installed — rather than something changed during a device's operational life. It **cannot be undone**.

Turning it on

Linux provides a set of commands for manipulating the eMMC partitions in the `mmc-utils` package. Install a standard Linux OS to the Compute Module device and install the tools as follows:

```
Code
sudo apt install mmc-utils
```

To get information about the eMMC (this command pipes into `less` as there is quite a lot of information to display):

```
Code
sudo mmc extcsd read /dev/mmcblk0 | less
```

Warning

The following operations are one-time — you can run them once and they cannot be undone. You should also run them before the Compute Module has been used, as they will incur data loss. The capacity of the eMMC will be significantly reduced compared to what it was previously (see 'Implementation details' above for the trade-off).

The command used to turn on pSLC is `mmc enh_area_set`, which requires several parameters that tell it over how much memory area the pSLC is to be enabled. The following example uses the entire area. Please refer to the `mmc` command help (`man mmc`) for details on how to use a subset of the eMMC.

Code

```
# Find out how big the eMMC is
MAXSIZEKB=$(sudo mmc extcsd read /dev/mmcblk0 | grep MAX_ENH_SIZE_MULT -A 1 | grep -o '[0-9]\+ ')
# Set the whole area to be pSLC
sudo mmc enh_area set -y 0 $MAXSIZEKB /dev/mmcblk0
reboot -f
```

After the device reboots, you will need to **reinstall the operating system**, as enabling pSLC will erase the contents of the eMMC.

The Raspberry Pi CM Provisioner software has an option to set pSLC during the provisioning process. This can be found on GitHub at https://github.com/raspberrypi/cmprovision.

Off-device file systems / network booting

Raspberry Pi computers are able to boot over a network connection, for example using the Network File System (NFS). This means that once the device has completed its first-stage boot, instead of loading its kernel and root file system from the SD card, it loads them from a network server. Once running, all file operations act on the server and not the local SD card, which plays no further role in the proceedings.

Cloud solutions

Nowadays, many office tasks take place in the browser, with all data being stored online in the cloud. Keeping data off local media storage obviously improves reliability, at the expense of needing an always-on connection to the internet, as well as possible subscription charges from cloud providers.

A full-blown Raspberry Pi OS installation, with the Raspberry Pi–optimised browser, can be used to access any of the cloud services from suppliers such as Google, Microsoft, Amazon, etc. An alternative is one of the thin-client providers, which replace Raspberry Pi OS with an OS/application that runs from resources stored on a central server instead of local storage. Thin clients work by connecting remotely to a server-based computing environment where applications and data are stored.

Application-level data integrity

Everything covered so far operates at the OS, mount, or file system level. For applications performing their own data storage or monitoring, the single most important, and most commonly overlooked, mitigation happens at the application level: no mount option or file system choice can protect a file that is overwritten in place.

If an application opens an existing file and writes new data directly into it, for example to update a configuration file or record the latest sensor reading, a corruption event during that write can leave the file partially written — neither the old version nor the new one — regardless of which file system or mount options are in use underneath it.

The standard, file-system-agnostic pattern to avoid this is:

1. Write the new data to a **temporary file** in the same directory as the target file, so that it ends up on the same file system (see below).
2. Call `fsync()` on the temporary file to force its contents to be physically written to storage, rather than left sitting in a write cache.
3. **Atomically rename** the temporary file over the target file, using `rename()`.

Code

```
int fd = open("data.tmp", O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd, buffer, length);
fsync(fd);                      // Force data to storage before the rename
close(fd);
rename("data.tmp", "data.dat");   // Atomic - data.dat is never partially written
```

This works because, within a single file system, `rename()` is **atomic**: a corruption event during the rename can only ever result in either the old file or the new file being present, never a partially written one.

Note

The temporary file must be created on the **same file system** as the target file. `rename()` is only atomic within a single file system — renaming across file systems (for example, from `/tmp` to a different partition) is implemented as a copy followed by a delete, which reintroduces exactly the failure mode this pattern is meant to avoid.

Warning

The `fsync()` call is not optional. Without it, the temporary file's data may still be sitting in the write cache at the point the rename happens, so a corruption event immediately afterwards can still leave the target file empty or containing old data, despite the rename having completed. Many higher-level 'atomic write' library functions perform the rename step but omit the `fsync()` call — check the documentation of any library you use for this before relying on it.

Conclusions

When the correct shutdown procedures are followed, the local storage media of a Raspberry Pi is extremely reliable. This works well in the home or in an office environment, where shutdown can be controlled, but when using Raspberry Pi devices in industrial use cases or in areas with an unreliable power supply, extra precautions can improve reliability.

In short, the options for improving reliability can be listed as follows:

- For SD cards specifically, use a well-known, reliable brand.
- Use a sensible journaling configuration — `data=ordered`, default write barriers, and `metadata_csum` enabled — and a commit time that reflects your tolerance for data loss against flash wear.
- Reduce writes using temporary file systems, or similar.
- Use a read-only rootfs where possible, ideally a native read-only image (SquashFS or EROFS) combined with an overlayfs, rather than an ext4 partition simply marked read-only, with a small dedicated read-write partition for any data that needs to persist.
- Protect the `/boot` partition, not just the rootfs.
- Use off-device storage such as network boot or cloud storage.
- Protect individual application files using write-to-temp-file, `fsync()`, and atomic rename, regardless of which file system or mount options are in use underneath.
- Use a UPS.

Contact us for more information

Please contact applications@raspberrypi.com if you have any queries about this white paper.

Web: www.raspberrypi.com

Raspberry Pi

Raspberry Pi Ltd