

EMBEDDED LINUX CONFERENCE EUROPE 2025

FAIL-SAFE EMBEDDED LINUX: DESIGNING FOR POWER RESILIENCE

Sergio Prado, Embedded Labworks
<https://www.linkedin.com/in/sprado>



WHOAMI

- 28+ years of experience in software development for embedded systems (Embedded Linux, Embedded Android, RTOS, etc).
- Consultant & Trainer at Embedded Labworks, providing consulting, training and software development services for embedded systems (software architecture, operating systems, BSP development, security).
<https://e-labworks.com/en>
- Free software contributor (Buildroot, Yocto Project, Linux kernel, etc).
- Blog and social media:
<https://sergioprado.blog>
<https://www.linkedin.com/in/sprado>
<https://twitter.com/sergioprado>
<https://www.youtube.com/spradotube>



WHAT CAN HAPPEN IN A POWER CUT?

- If power is cut from an embedded Linux device, any in-progress operation stops immediately!
- Some peripherals (e.g., MCUs, FPGAs) might require proper shutdown sequences, or otherwise they can enter into undefined states and need manual reset or reprogramming.
- If the system controls moving parts (motors, actuators), a power cut can leave them in unsafe positions.
- Sudden power loss (or restoration) can cause voltage transients that may damage hardware.
- You may lose data or corrupt a filesystem.



FILESYSTEM CORRUPTION

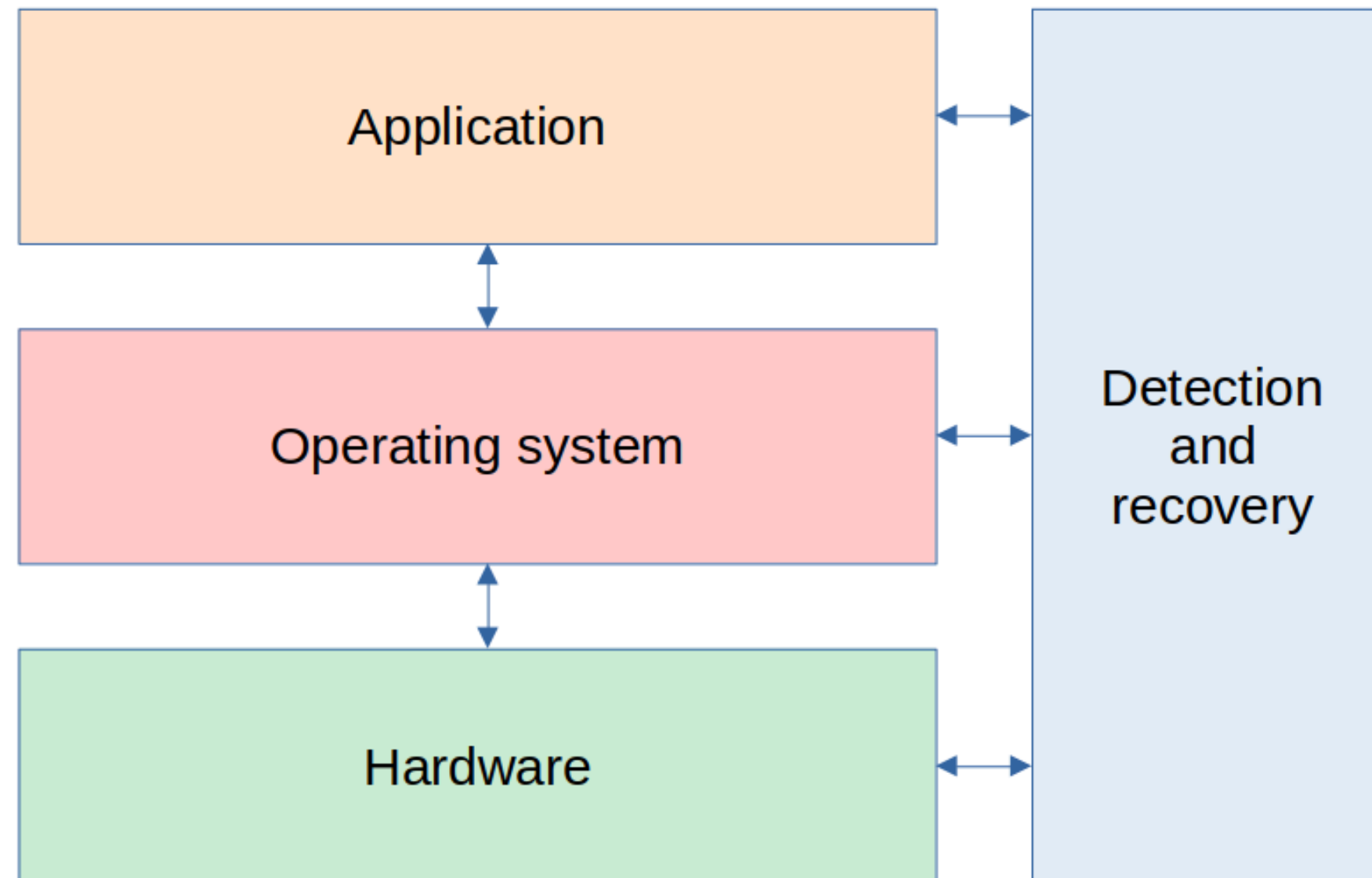
- The focus of this talk is on minimizing data loss and preventing filesystem corruption on power cuts.
 - Most techniques we will learn can also benefit other scenarios.
 - Other factors that can cause filesystem corruption are outside the scope here.
- Also, it is important to mention that improving reliability comes with trade-offs:
 - Additional hardware costs.
 - Reduced performance.
 - Increased write amplification and flash wear.
 - Increased development time and maintenance complexity.



HOW TO DESIGN FOR POWER CUT RESILIENCE?

- Reduce the probability of a failure to happen.
 - Different mitigations can be implemented in a layered approach:
 - Hardware level.
 - Operating system level.
 - Application level.
- No system is completely immune to power-cut issues, so when a failure happens, have mechanisms to **detect** and **recover** from it.

A LAYERED APPROACH

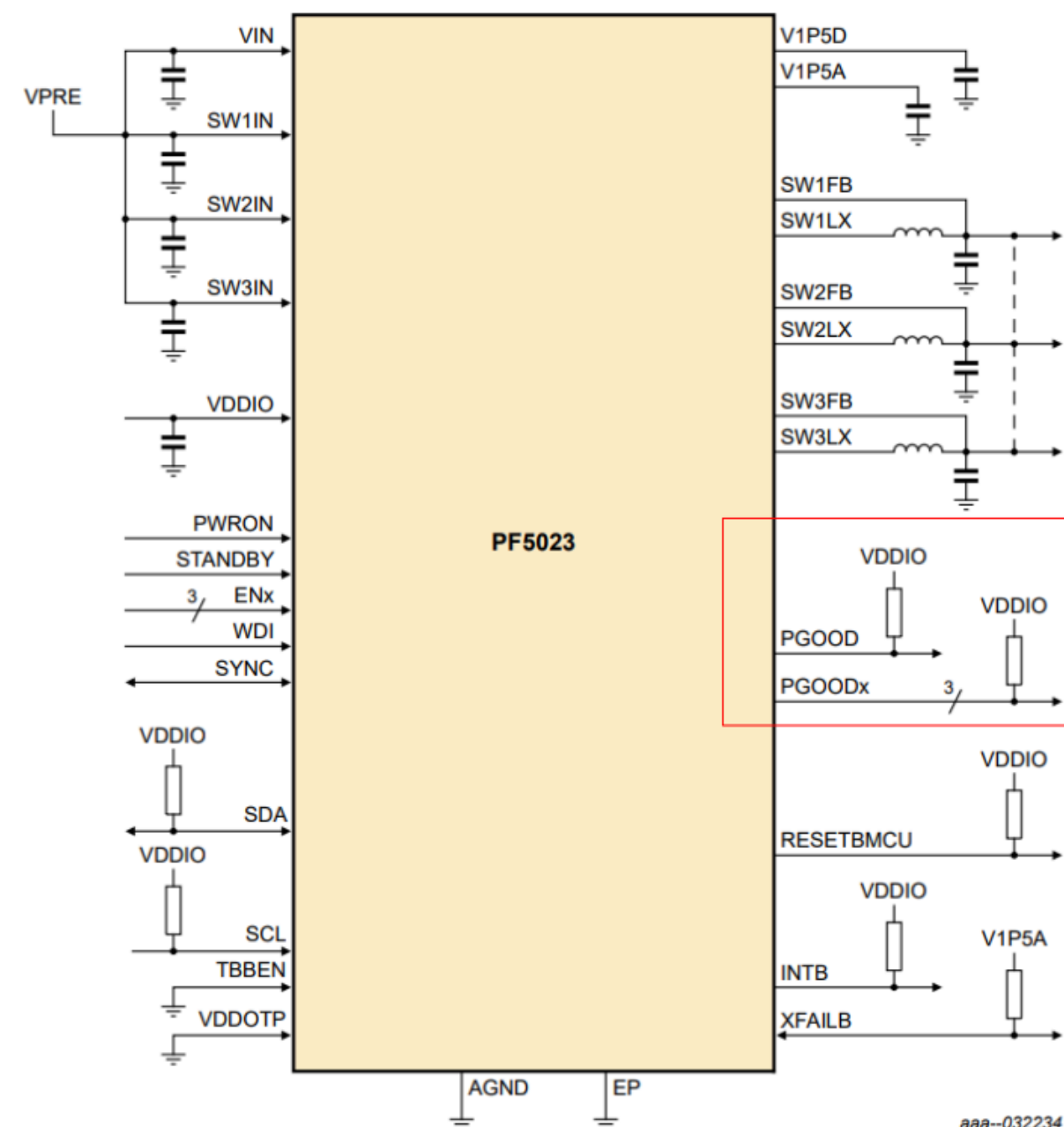


HARDWARE-LEVEL MITIGATIONS

- Make sure the power source provides input protection and guarantee power integrity and prevention against spikes.
- Add a small battery or supercapacitor to hold up power for some time.
- Use a PMIC to monitor the power supply and trigger a protective action if the voltage drops below a certain threshold (brownout detection).
 - When power loss is detected, assert a GPIO to the OS so it can perform an orderly shutdown.
- Prefer industrial-grade eMMC/SD with power-loss protection (ECC, internal capacitors, etc) and enhanced area/reliable write features.



NXP PMIC PF5023



SUPERCAPACITOR EXAMPLE

Analog devices - Design Note 450

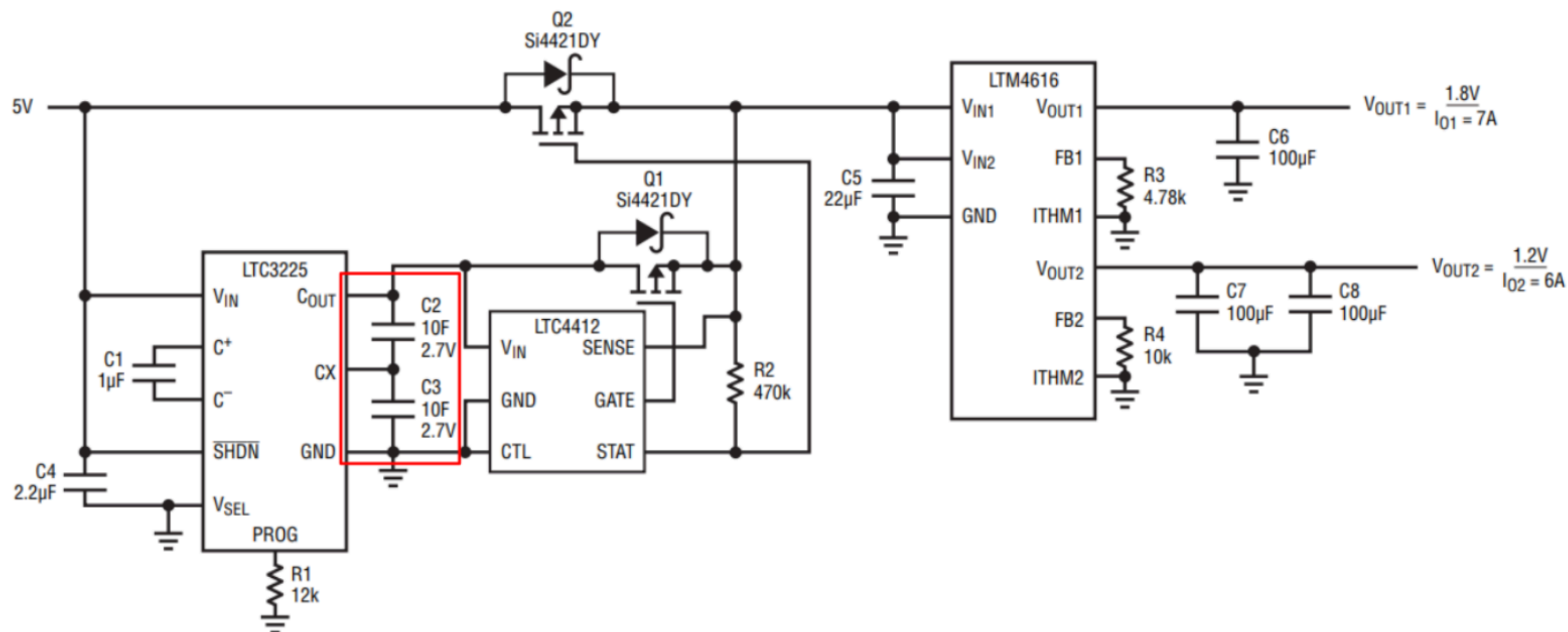


Figure 1. 5V Ride-Through Application Circuit Delivers 20W for 1.42 seconds

EMMC POWER PROTECTION

SMARTsemi Industrial Grade eMMC eMMC Datasheet

- Supports Write protect, Secure Write Protection
 - Supports Cache, Cache Barrier, Cache Flushing Report
 - Hardware/ Software Reset
 - Supports PON, Sleep/Awake
- Robust Data Protection and Endurance
 - Configurable ECC engine with zero overhead pipeline greatly reduces data loss rates and increases data endurance
 - Static data refresh and early block failure monitoring/retirement ensure the data reliability
 - Power loss detection and mapping table auto-rebuild algorithm support power-down data protection
 - Global wear leveling maximizes product lifespan with minimal wear leveling and write amplification overhead

EMMC DEVICE CACHE

- eMMCs might have a write cache, a fast and temporary memory inside the storage device used to stage data before committing it to the slow, non-volatile media (e.g. NAND memory).
- It improves performance and helps reduce wear on flash memory, but it can result in data loss or corruption if power is removed before the data is committed.
- The kernel issues flush commands (CACHE_FLUSH command as defined in the JEDEC standard) when an application calls *fsync()*, performs a disk sync, or unmounts a filesystem.
- The device cache can be disabled via a control file at `/sys`:

```
# cat /sys/class/block/mmcblk0/queue/write_cache  
write back  
# echo "write through" > /sys/class/block/mmcblk0/queue/write_cache
```



ENHANCED AREA AND WRITE RELIABILITY

- Enhanced area and write reliability are two additional features provided by recent eMMCs to improve the robustness of its storage features.
 - Enhanced area improves endurance, retention and reliability; usually by storing data in pSLC mode (e.g. using just one bit per cell instead of 2 on an MLC NAND).
 - Write reliability ensures that a write operation is atomic and safe, usually by internally duplicating a write into two separate physical locations.
- Be aware that the implementations details might depend on the eMMC.
- Enabling those features can be done with the *mmc* command from the *mmc-utils* package, and are usually irreversible!



KINGSTON EMMC DATASHEET



Configurations

Kingston's e•MMC™ products support a variety of configurations that allow the e•MMC™ device to be tailored to your specific application needs. The most popular configurations described below are each offered under standard part numbers.

Standard MLC – By default the e•MMC™ device is configured with the NAND flash in a standard MLC mode. This configuration provides reasonable performance and reliability for many applications.

Pseudo Single Level Cell (pSLC) – The MLC NAND flash in the Kingston e•MMC™ device can be configured to further improve device endurance, data retention, reliability and performance over the standard MLC configuration. This is done by converting the NAND MLC cells to a pseudo single level cell (SLC) configuration. In this configuration, along with the performance and reliability gains, the device capacity is reduced by 50%. This one-time configuration is achieved by setting the e•MMC™ enhanced attribute for the hardware partition.

Enhanced Reliable Write – When not configured as pSLC, MLC NAND flash stores 2 bits of information in 4 energy levels per NAND flash cell. Since these paired bits are organized in different NAND pages, there is a possibility that a power failure while programming a page could corrupt a paired page that was already programmed. For the Kingston e•MMC™, this condition is rare and the possibility is further reduced due to the device's built-in data protection with on-board error correction code (ECC) bits. With *reliable write* set, the onboard e•MMC™ controller will back-up any paired pages to ensure that there is no data loss during sudden power failure. This configuration can result in a write performance penalty of up to 20% over the standard MLC configuration.



MMC COMMAND

```
$ mmc --help
...
mmc enh_area set <-y|-n|-c> <start KiB> <length KiB> <device>
  Enable the enhanced user area for the <device>.
  Dry-run only unless -y or -c is passed.
  Use -c if more partitioning settings are still to come.
  NOTE! This is a one-time programmable (unreversible) change.

mmc write_reliability set <-y|-n|-c> <partition> <device>
  Enable write reliability per partition for the <device>.
  Dry-run only unless -y or -c is passed.
  Use -c if more partitioning settings are still to come.
  NOTE! This is a one-time programmable (unreversible) change.
...
```



OS-LEVEL MITIGATIONS

- At the operating system level, different techniques can be applied to mitigate issues with power cuts, including:
 - Define a robust partition layout scheme.
 - Select an appropriate filesystem for the storage media in use.
 - Use mount options tuned for power resilience.
 - Have a safe and reliable update mechanism.

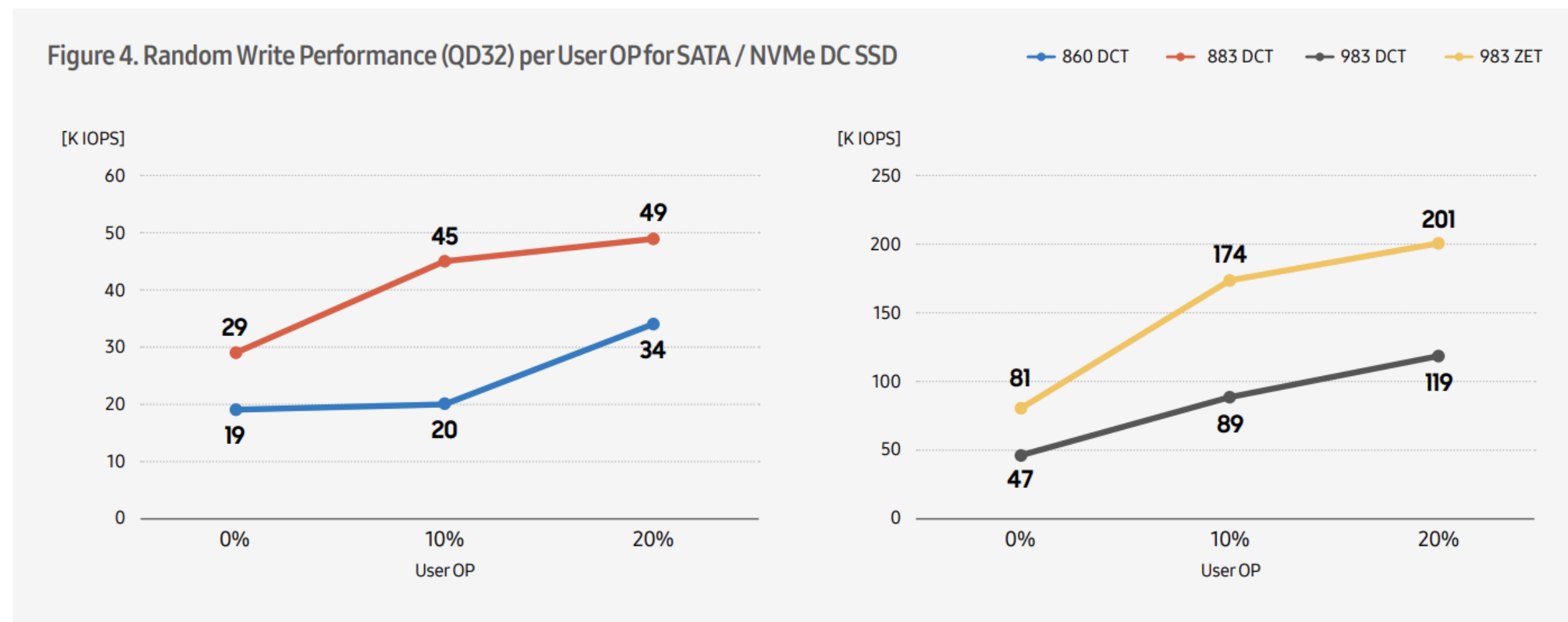


GOOD PRACTICES FOR PARTITION LAYOUT

- Separate read-only and read-write data into different partitions.
- For read-write data:
 - Have volatile and persistent data at different partitions/locations.
 - For persistent data, separate critical data (e.g. configs and databases) and non-critical data (e.g. logs) into different partitions.
 - Duplicate partitions for recovery whenever possible (A/B system).
- On flash-based devices with an FTL (eMMC, SD card, SSD, etc), leaving 10~20% of the space unpartitioned might improve both endurance and performance by providing the FTL with additional free blocks to manage.
 - This is usually called **overprovisioning**.



OVERPROVISIONING



Source: [Over-Provisioning Benefits for Samsung Data Center SSDs](#)

READ-ONLY ROOTFS

- Mitigates the risk of a sudden power loss from corrupting essential OS files, which would prevent the system from booting properly.
- Both SquashFS and EROFS are good filesystem options for a read-only partition.
 - The decision between SquashFS and EROFS usually depends on the read performance versus the compression rate tradeoff.
 - Using a writable filesystem like ext4 in read-only mode is also an option, but it opens the possibility for the partition to be remounted read-write.
- In case writing to the rootfs is needed (e.g. `/etc`), there are two common approaches:
 - Create links to a writable partition.
 - Use an overlay filesystem like OverlayFS.



DATA PARTITION

- For storing runtime data, there are some filesystems options, depending on the storage technology used.
- For conventional hard-disks (not common on embedded), ext4 and Btrfs are both good options.
- For hybrid devices that have a NAND flash and an FTL (eMMC, SD cards, etc), both ext4 and F2FS are good options.
 - Btrfs is also an possibility, but not common yet on embedded use cases.
- For MTD devices (raw NAND/NOR), UBI/UBIFS is the default option.
 - JFFS2 is also an alternative, but mainly for very small flashes.

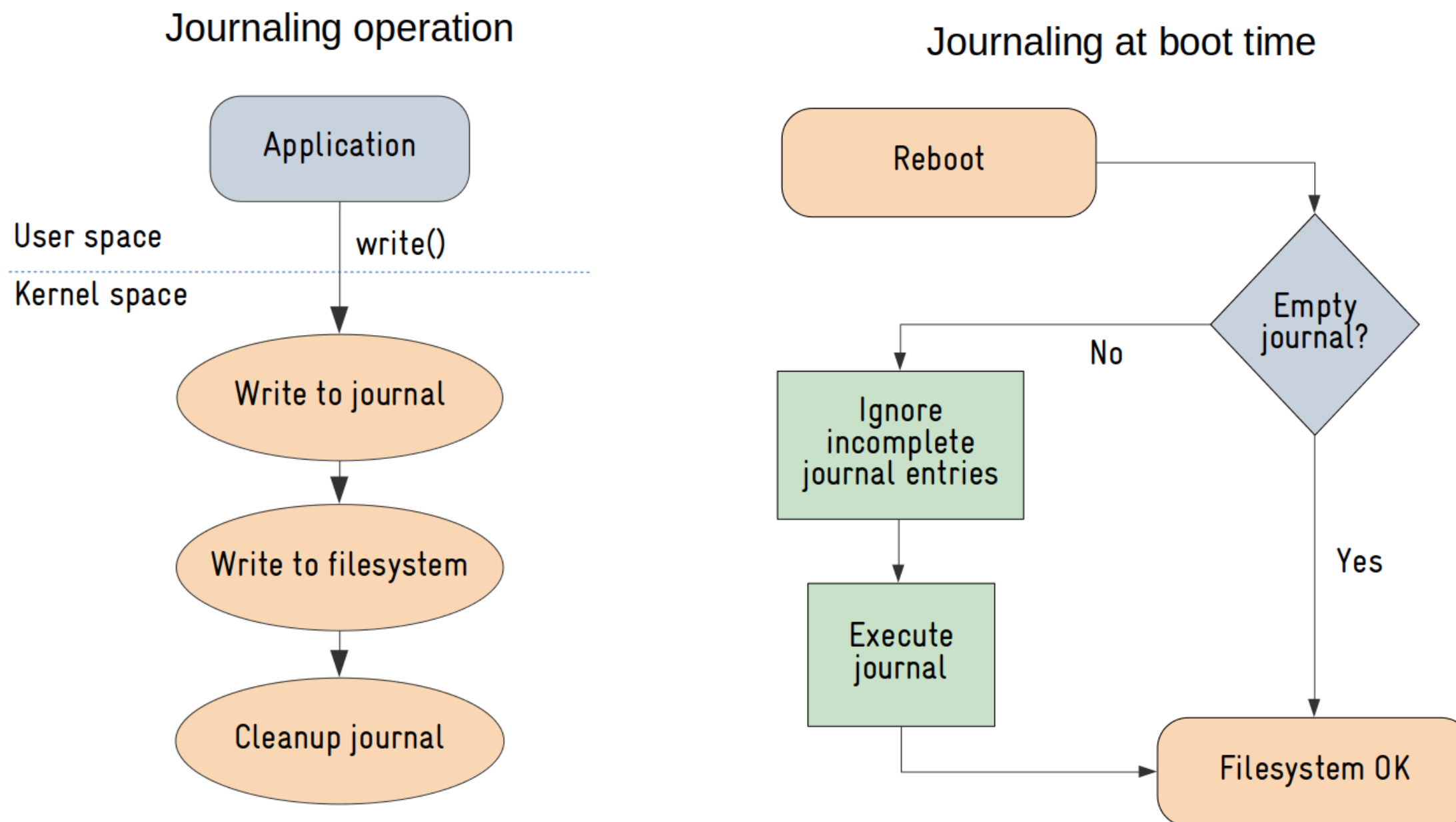


EXT4

- ext4 is still one of the most commonly used filesystems on embedded Linux.
- It is a journaling filesystem, which means it keeps a journal (a circular log) on disk that records pending data and metadata updates before applying them to the filesystem.
 - This ensures data and metadata consistency after a crash, as the journal can be replayed when mounting the filesystem or roll back incomplete updates.
- The journal itself can be stored on the same partition (default) or in a separate block device.



HOW DOES JOURNALING WORK?



JOURNALING MODES

- The ext4 filesystem supports three journaling modes, configured via the *data=* mount option:
 - **ordered**: metadata only, though ext4 ensures all data blocks for a write are flushed to disk before journaling the related metadata.
 - **writeback**: metadata only with no ordering, which means metadata can be committed before data blocks are written.
 - **journal**: both metadata and data are written to the journal before committing to the filesystem.
- The journal mode is the safest for both metadata and data integrity, and the best against partial writes, but it is also the slowest and has increased write overhead.



EXT4 MOUNT OPTIONS

Option	Description
<i>noatime</i>	Do not update inode access times
<i>barrier=1</i>	Enforce proper ordering of journal commits, making volatile disk write caches safe to use, at some performance penalty
<i>sync</i>	Makes all writes synchronous, which provides maximum safety but has a significant performance impact
<i>commit=X</i>	Where X is the time (in seconds) that data can sit in kernel memory before being committed to the disk (default 5)
<i>data=journal</i>	Journal both metadata and data, which doubles write traffic but provides the strongest integrity guarantees
<i>journal_checksum</i>	Enable journal checksums for corruption detection



F2FS

- F2FS (Flash-Friendly File System) was designed by Samsung specifically for NAND based devices with an FTL (eMMC, SD cards, SSD, etc).
- It has a log-structured design, where writes are sequentially appended in segments instead of overwriting in place.
 - Compared to ext4, it avoids write amplification from in-place updates and better aligns writes to erase-block boundaries.
- It has a garbage collector (also called cleaning process) that, from time to time, reclaims segments with obsolete data.
- It has also a checkpointing system (snapshots of filesystem metadata) to recover after a crash (it keeps two checkpoints, and when mounting the filesystem, it chooses the most recent and valid one).



F2FS MOUNT OPTIONS

Option	Description
<i>noatime</i>	Do not update inode access times
<i>barrier</i>	Preserve write ordering by issuing <i>cache_flush</i> commands, protecting against cache reordering
<i>data_flush</i>	Enables the filesystem to periodically flush user data to the storage device before a checkpoint operation
<i>fsync_mode=strict</i>	When an application calls <code>fsync()</code> , flush both data and metadata to the device
<i>checkpoint=enable</i>	Always maintain a recent, valid checkpoint for crash recovery



F2FS KNOBS

- Increase GC idle interval to avoid the garbage collector during critical disk writes:

```
# echo 3600000 > /sys/fs/f2fs/<disk>/gc_idle_interval
```

- Force an immediate checkpoint:

```
# echo 1 > /sys/fs/f2fs/<disk>/cp_interval
```

- Show checkpoint status in real time:

```
# cat /sys/fs/f2fs/<disk>/stat/cp_status
```

- For more information, see the [Linux kernel documentation](#).



MTD AND UBI

- Managing NAND devices without an FTL (Flash Translation Layer) is much more work because one have to deal (in software) with erase-before-write operations, bad block management, wear leveling, etc!
- In the Linux kernel, flash memory is managed by a layer called MTD (Memory Technology Devices), a standard subsystem for drivers to expose an interface to flash memory devices (NAND, NOR, etc).
- UBI (Unsorted Block Images) is a volume management layer that sits between MTD devices and the filesystem.
 - Implements wear-leveling and tracks bad blocks.
 - Maps logical erase blocks (LEBs) into physical erase blocks (PEBs).
 - Provides runtime volume management (create/resize/remove volumes).



UBIFS

- UBIFS is a filesystem designed to run on top of UBI volumes, optimized for raw flash memory.
- It has a log-structured design, appending changes instead of overwriting, reducing flash stress and making updates safer.
- It is designed to be **tolerant of power failures**, using a journal to ensure on-flash consistency.
 - When a power cut occurs, it can replay the journal to restore the filesystem to a consistent state.
- UBIFS checksums everything it writes to guarantee data integrity (metadata checksummed by default; data checksums via mount option).



UBIFS MOUNT OPTIONS

Option	Description
<i>noatime</i>	Do not update inode access times
<i>sync</i>	All writes committed immediately - safest but slower and more write-heavy
<i>chk_data_crc</i>	Enables data CRC checking



VOLATILE DATA

- When dealing with volatile data, it is always a good practice to avoid unnecessary writes to flash to reduce wear and the risk of corruption from power cuts.
- tmpfs is a filesystem that stores its contents in volatile RAM, and it is usually mounted in places like */tmp*, */run* and */var/run*.
- When using tmpfs, reads/writes are as fast as RAM and the data disappears at reboot or power loss.
- Can be used for temporary files, runtime state (sockets, PID files, etc), ephemeral logs, and so on.



DUPLICATING PARTITIONS

- Duplicating partitions in an dual-slot scheme (A/B) is another strategy to make the operating system more resilient against corruption.
- The device has two identical rootfs partitions (A and B) and the bootloader knows which slot to boot from (via an environment variable, boot flag, etc).
- If the boot fails from one partition, then the bootloader might try to boot from the other partition.
 - You need to make sure the system doesn't get stuck at boot time (reboot on panic, watchdog, etc).
- Also useful as part of a (remote) update system (Mender, RAUC, SWUpdate, etc).



APPLICATION-LEVEL MITIGATIONS

- At the application level, different techniques can be applied to make the software tolerant to sudden power loss.
- In a nutshell, you need to a) reduce the "vulnerability window" and b) make sure writes are atomic.
- To reduce the "vulnerability window", one can open files with the `O_SYNC` flag and/or call `fsync()` when writing critical data.
- Writing atomically to disk can be achieved by following the pattern:
 - Write the new content to a temporary file.
 - `fsync()` the temporary file to ensure data is on persistent storage.
 - Rename the temporary file over the old one.



MORE APPLICATION-LEVEL MITIGATIONS (1)

- Minimize writes!
 - For example, avoid keeping operation logs in a persistent partition. If persistent logs are required, do it on a separate partition.
- Avoid constant rewrites of the same file.
 - For example, one can buffer changes in memory or in a tmpfs mountpoint and periodically sync to persistent storage.
- Make sure external software supports atomic updates to disk.
 - For example, use SQLite in WAL (Write-Ahead Logging) journal mode to guarantee atomic commits.



MORE APPLICATION-LEVEL MITIGATIONS (2)

- Include checksums or hashes with your stored data.
 - Verify them on load to detect corruption early and fall back to a known-good state.
- Keep redundant copies of critical files.
 - Keep two (or more) rotated copies of critical configs; on startup, pick the latest valid, fall back if corrupt.
- Implement a SIGTERM handler and properly handle shutdown notifications.
- Whenever possible, implement a shutdown hook.



SHUTDOWN HOOK

- If power is lost but the hardware provides a short hold-up time, the operating system can take action before the system shuts down completely, for example, flushing pending writes, sending a power-off command to the eMMC, and notifying processes.
- A typical setup is a PMIC that raises a GPIO on power-cut events, allowing applications to begin a controlled shutdown before power is fully gone.
- This can also be integrated with systemd by mapping the GPIO to generate a *KEY_POWER* event, and configuring systemd to initiate the shutdown when the event is received.
- Some handling can also be done in kernel space, for example by registering for the *REGULATOR_EVENT_UNDER_VOLTAGE* event via the regulator consumer driver API.



MONITOR UNINTENDED REBOOTS

- Identify unintended reboots and take some action when it happens.
 - Create an "I'm running" marker and delete it at clear shutdowns.
 - If the marker is present on the next boot, then there was an unintended reboot.
- When an unintended reboot is identified, the system can enter into "recovery mode", and different actions can be taken, including:
 - Inspect kernel logs for errors.
 - Check the status of mountpoints.
 - Run tools like *fsck*, especially on writable partitions.
 - Check the storage device health (e.g. via *mmc* command on eMMC).



RECOVER FROM CORRUPTION

- If the issue is at the OS level (bootloader, kernel, etc), redundancy is usually the best recovery option (e.g. dual A/B partition).
 - Initramfs can be used to check the storage before pivoting to the real rootfs.
 - A watchdog might be useful to recover from kernel/early-boot failures.
- If the OS boots and the issue is at the application level, the solution depends on the use case. Examples:
 - Restore configs from backup if main copy is corrupt.
 - Try to recover the partition with *fsck*.
 - As a last resort, clean/format the partition (i.e. factory reset).



CHECKLIST FOR POWER RESILIENCE

- Design the hardware carefully (input protection, alternative power source, PMIC, storage device selection, etc).
- Avoid writes where possible (rootfs read-only, tmpfs).
- Separate critical and non-critical data into different partitions.
- Select an appropriate filesystem and mount it with optimal flags.
- Make sure to always do atomic writes to the storage.
- Keep the "vulnerability window" very short (e.g. use *fsync()* for critical data).
- Implement detection and automated recovery/rollback.
- And of course, don't forget the tradeoffs!



EMBEDDED LINUX CONFERENCE EUROPE 2025

THANK YOU! QUESTIONS?

Sergio Prado, Embedded Labworks
sergio.prado@e-labworks.com
<https://www.linkedin.com/in/sprado>
<https://sergioprado.blog>

