System Updates with NuttX Bootloader

NuttX Workshop 2025

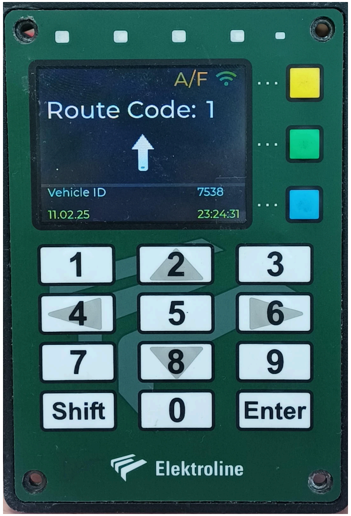
17. 10. 2025

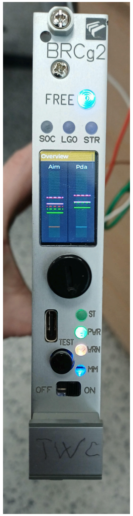
Michal Lenc, Elektroline a.s.

Use Case and Requirements

- NuttX embedded devices placed in panel boards, trolley buses, trams…
- devices typically connected over CAN bus with a central Linux based device
- large image – hundreds of kBs
- external NOR memory available
- requirements:
    - remote firmware update (basically over-the-air)
    - new firmware uploaded during standard device operation (A/B update)
    - update performed after device reset
    - explicit new firmware confirmation
    - old firmware is recovered during next boot if new not confirmed
    - correct behavior even if power outage occurs during update
    - short update (reboot) time
        - need to avoid long writes and erases in external NOR memory

Use Case and Requirements (ii)





NuttX and MCUboot

- MCUboot is an open source secure bootloader
  - https://docs.mcuboot.com/
- use widely by Zephyr, but also provides support for NuttX
- NuttX support based on BCH and FTL layer with erase page buffering
- new image upload is the responsibility of the application
- bootloader just performs swap during reboot if new image is detected
- new image confirmation and revert posibility
- has XIP mode, RAM load, copy only mode and several swap algorithms
  - swap using scratch
  - swap using offset
  - swap using move

Swap Algorithms

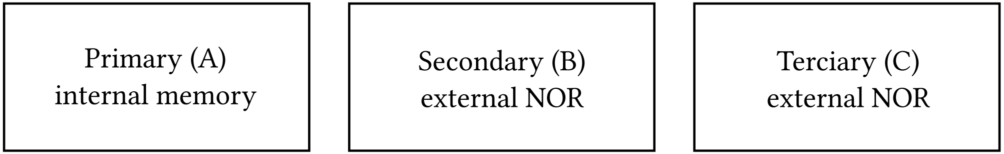
- two partitions (primary and secondary)
- image runs from the primary partition, secondary is for new image and recovery
- update process
  - application service uploads new image to secondary
  - primary and secondary swap during update process
  - needs to remember swap state to recover if power outage occurs
- scratch algorithm has third partition that stores copied sectors
  - this partition has large wear
  - marked as potentially deprecated in the documentation
- offset and move algorithms has a bit larger secondary and moves empty space
- still not ideal...

Swap Algorithms (ii)

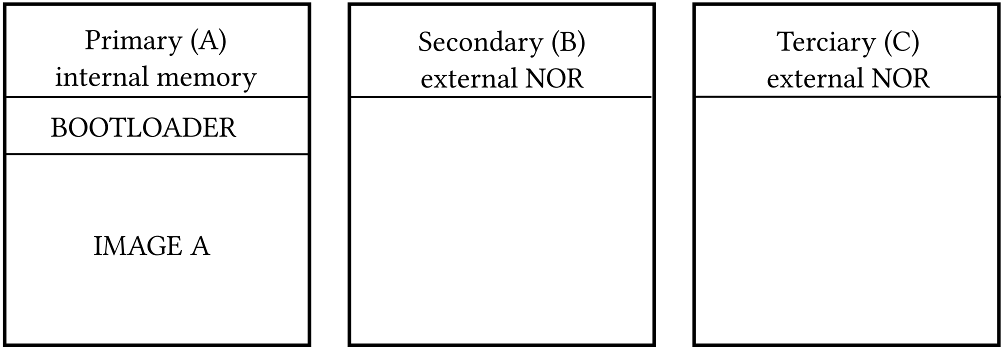
- these algorithms have to create recovery during every update
  - significant bottleneck if secondary partition is on external memory (slow writes and erases)
  - causes larger partition wear
- update progress kept in image tail
  - this tail is located at the end of the partition -> image padded to partition size
  - this slows update process (larger image to upload and swap) even if firmware is small
- image confirm information stored as a flag in the tail
- image intended for direct programmer flashing marked as stable -> diferent FWs for system and update
- need to write to internal flash from which FW runs to confirm

Alternative

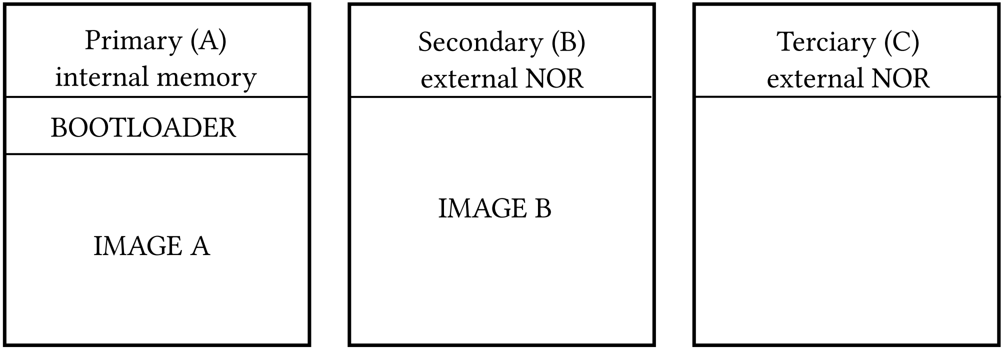
- the biggest issue are write and erase operations on NOR
- recovery should be always present to avoid them
  - recovery image is basically the previous update image that was confirmed
- impossible with two partitions, needs three
- larger memory requirements, but faster and more considerate update



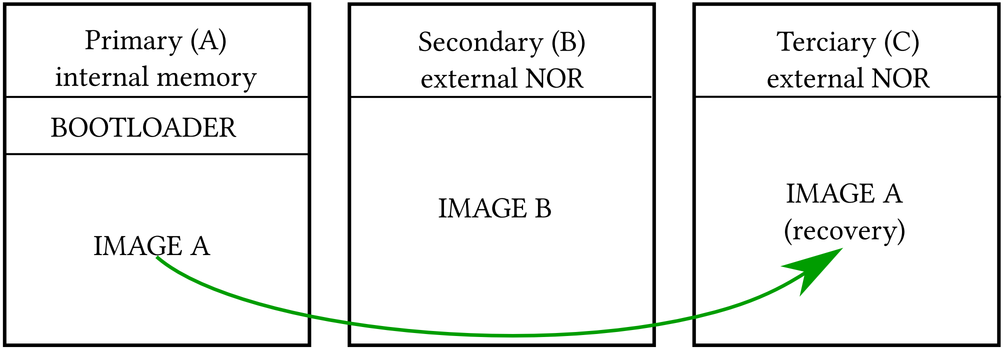
Alternative (ii)



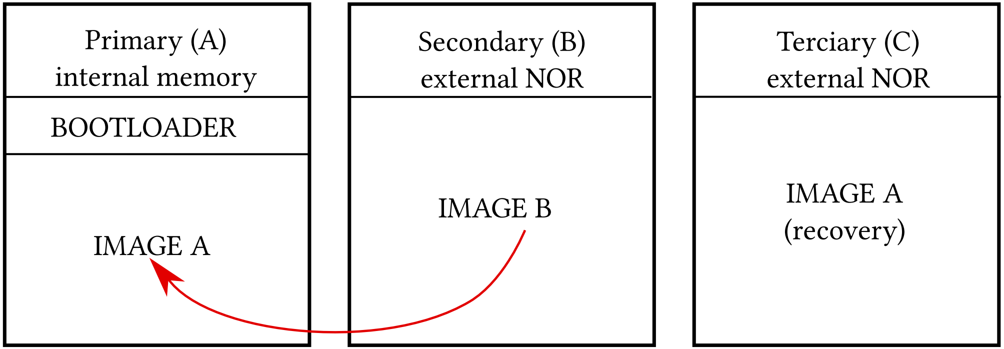
Alternative (iii)



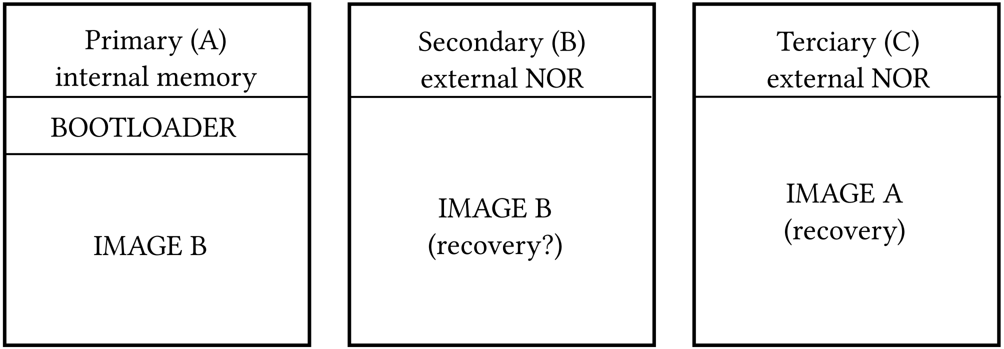
Alternative (iv)



Alternative (v)



Alternative (vi)



Alternative (vii)

- recovery is basically created during firmware upload
  - this may take time, but the device is still in standard operation mode
  - the only exception is the first update after the device is flashed with the programmer
- update process only rewrites the internal memory
  - significantly faster as we avoid writes and erases on slow external NOR
  - less wear
- update time sped up from aproximately 5 minutes (with scratch algorithm) to 20-30 seconds for 2 MB image
- still present issues:
  - image has padding to partition size
  - confirmation done by write to program memory
  - needs three partitions, 50 % more reserved memory
- algorithm proposed to MCUboot but merge request was declined

How is bootloader implemented?

It’s not that complicated if we skip cryptography.

- bootloader placed at the begining of primary partition
  - it is run every time after the boot and decides the operation it takes
- currently run image also in the primary with defined offset
- image has header (magic, size, ...), possibly a tail (flags)
- swapping has harder logic, but our algorithm just copies from A to B
- two layers
  - algorithm/decision logic
  - access to partitions over BCH/FTL/MTD layer

Why not to write the bootloader directly to NuttX?

1 NXBoot

Header

- magic – special value determining this is NXBoot image
- header version – ensures backwards compatibility
- header size – image is behind the header
- crc – calculated from the rest of the header and image
- image size
- identifier – unique number that matches bootloader identifier
  - this ensures image for different board is not accidently uploaded and run
- next header address – future header expansions
- image version – Semantic Versioning 2.0
  - not intended for NuttX version, but your firmware version

Header (ii)

```c
struct nxboot_img_header
{
  uint32_t magic;
  struct nxboot_hdr_version hdr_version;
  uint16_t header_size;
  uint32_t crc;
  uint32_t size;
  uint64_t identifier;
  uint32_t extd_hdr_ptr;
  struct nxboot_img_version img_version;
};
```

Header (iii)

```c
struct nxboot_img_version
{
  uint16_t major;
  uint16_t minor;
  uint16_t patch;
  char pre_release[NXBOOT_HEADER_PRERELEASE_MAXLEN];
};

struct nxboot_hdr_version
{
  uint8_t major;
  uint8_t minor;
};
```

Tail

- contains information about image stability (confirm or not), optinally other metadata
- flags should be padded to write page size to avoid erases during writes
- for three partitions we have to mark the image as already written to primary
  - it is not overwritten by the recovery and we have to avoid repeated updates
- do we actually have to use tail flag for it?
  - we can mark the image as already updated by erasing the first erase page
  - this erase page is copied back from primary during confirm
  - also used to mark image as stable -> if it has a backup, it is stable
- disadvantages:
  - first sector in B/C has more wear than the others
  - confirmation may take longer time due to entire erase page write

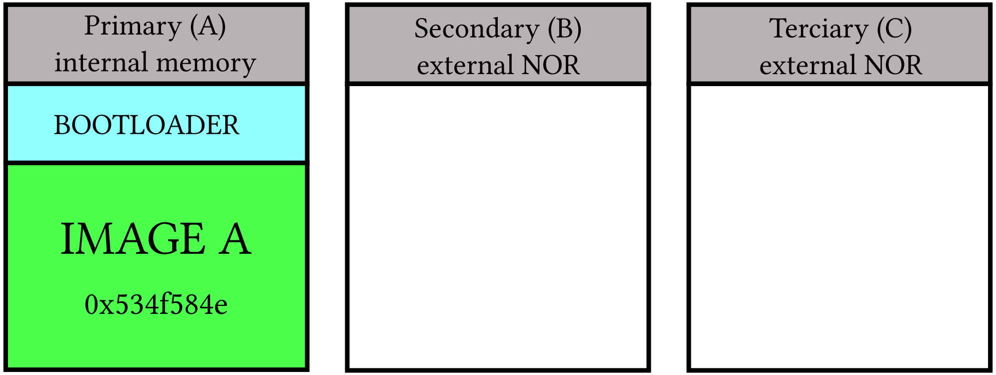
What if we upload new firmware with a programmer? It would be unstable…

Tail (ii)

- we have to separate directly flashed image from the one provided by the bootloader
- allow two magic values in image's header (standard and internal)
- image is build only with standard magic!
  - one image for initial programming and for updates
  - standard magic in primary partition -> **always stable**
- bootloader switches from standard to internal magic when copying new image to primary
- B/C partitions vice versa -> standard magic is update, internal recovery
- first two bits of internal magic used as a pointer to a recovery
  - needed to ensure correct update slot after first erase page is erased
  - simplifies and speeds up internal logic

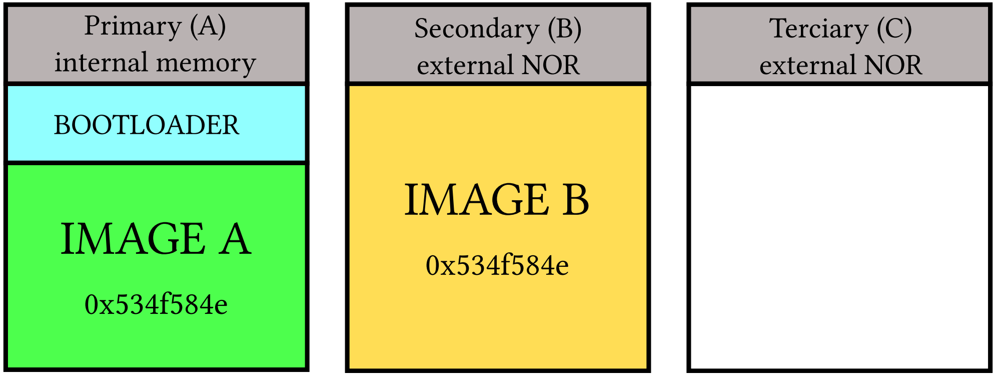
```
#define NXBOOT_HEADER_MAGIC     0x534f584e
#define NXBOOT_HEADER_MAGIC_INT 0xaca0abb0
```

NXBoot design – initial state



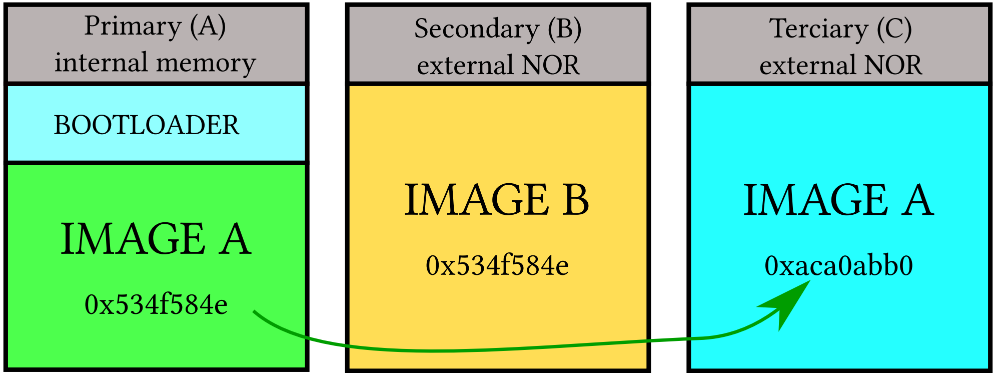
- image with normal magic written in A -> confirmed
- B/C empty

NXBoot design – update upload



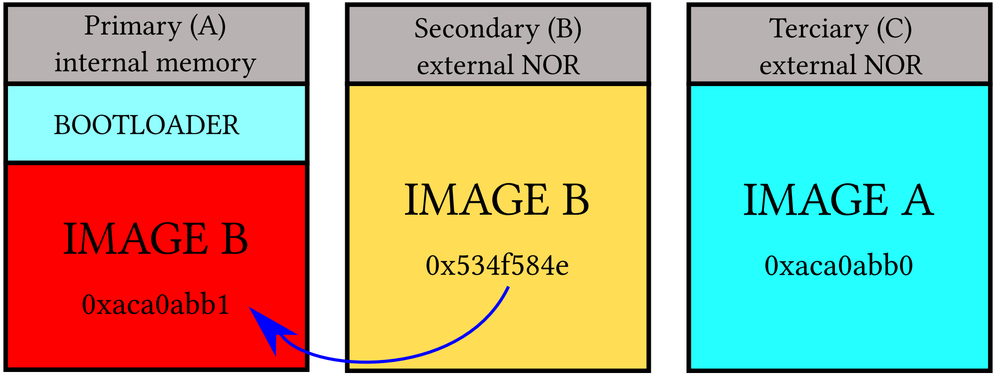
- image with normal magic written in A -> confirmed
- image with normal magic written in B -> update

NXBoot design – create recover



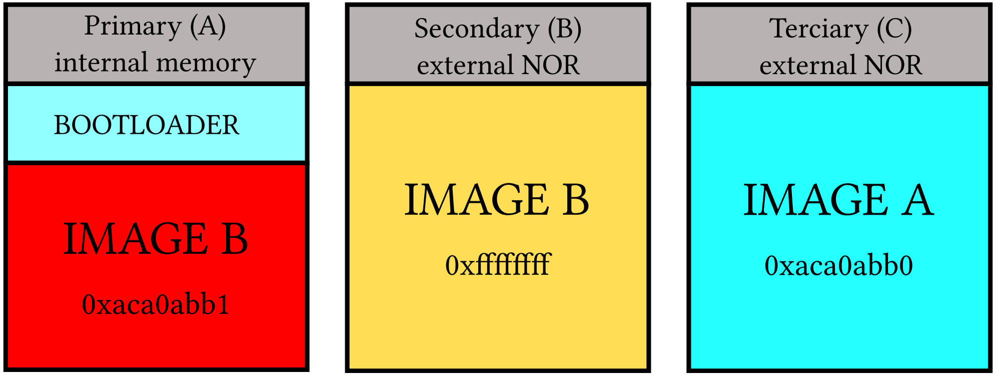
- recovery of A written to C with internal magic
- update is performed after recovery is succesfully written

NXBoot design – perform update



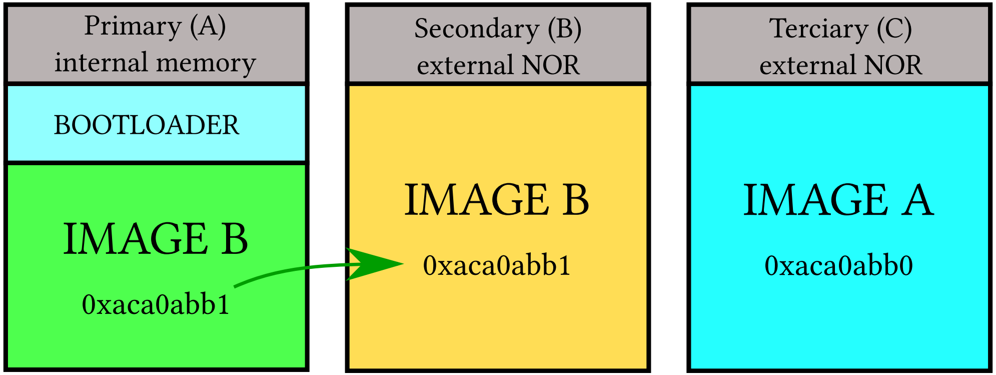
- copy image from B to A
- B should be marked as updated after copy is finished

NXBoot design – mark as updated



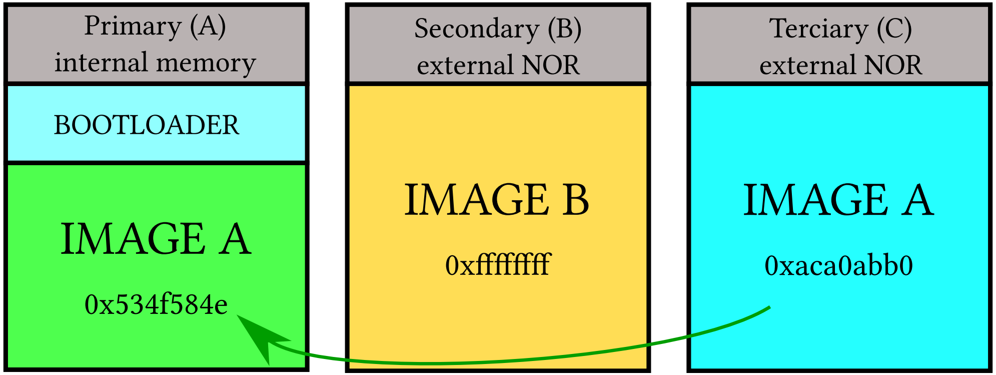
- remove first erase page of B (this include image's header)
- update is finished, not confirmed image in A boots

NXBoot design – confirmation



- first errase page from A is copied to B
- image has its recovery -> it is confirmed

NXBoot design – revert



- image copied from C to A with standard magic
- B is untouched, it will be used as a next update slot

2 NXBoot API

Configuration

```
CONFIG_BOOT_NXBOOT=y
CONFIG_NXBOOT_PRIMARY_SLOT_PATH="/dev/ota0"
CONFIG_NXBOOT_SECONDARY_SLOT_PATH="/dev/ota1"
CONFIG_NXBOOT_TERCIARY_SLOT_PATH="/dev/ota2"
```

Header size defines the offset where the actual image starts.

```
CONFIG_NXBOOT_HEADER_SIZE=0x200
```

Platform identifier is 64 bit number imprinted in the bootloader, images must match it.

```
CONFIG_NXBOOT_PLATFORM_IDENTIFIER=...
```

Configuration (ii)

- bootloader performs update/revert after every boot if available
- this operation can be limited to software reset only if target platform supports *BOARDCTL_RESET_CAUSE* ioctl

```
CONFIG_NXBOOT_SWRESET_ONLY=y
```

- if set, operation is performed only for following causes

```
BOARDIOC_RESETCAUSE_CPU_SOFT: software reset
BOARDIOC_RESETCAUSE_CPU_RWDT: watchdog error
BOARDIOC_RESETCAUSE_PIN: reset button
```

TLDR: update/revert is not performed for power outage, but bootloader still finishes the operation if interrupted by power shutdown.

Configuration (iii)

```
CONFIG_NXBOOT_PREVENT_DOWNGRADE=y
```

- ensures only update to newer version (per Semantic Versioning rules) is possible
- by default update is run for every version that differs or if image's CRC is not the same

WARNING: downgrade prevention now works only for MAJOR.MINOR.PATCH and ignores PRERELEASE.

```
CONFIG_NXBOOT_PROGRESS=y
CONFIG_NXBOOT_PRINTF_PROGRESS=y
```

- provides more verbose progress report to STDOUT, thanks to Tim Hardisty for the patch!

Configuration (iv)

- typical usecase is to build two images
  - bootloader
  - system image (*.nximg*)
- config option *CONFIG_NXBOOT_BOOTLOADER* is set for bootloader build.

```
CONFIG_NXBOOT_BOOTLOADER=y
```

- bootloader can be flashed to the board as standard NuttX image (bin, elf file, etc.).
- typically at the begining of program memory
- system image needs to prepend NXboot header and is flashed right after the bootloader
  - example location is *dev/ota0*, bootloader's location is nameless

Image creation

Python script that adds header at the image's begining.

```
python3 apps/boot/nxboot/tools/nximage.py \
  --version VERSION \
  --header_size CONFIG_NXBOOT_HEADER_SIZE \
  --identifier 0x0 \
  nuttx.bin image.nximg
```

- calculates CRC and adds requiered fields
- used for both update image and image flashed by programmer
- needs *semantic_version* and *argparse* modules

Application access

Opens the partition to which an update image should be stored and returns its file descriptor.

```
int nxboot_open_update_partition(void);
```

Function *nxboot_get_confirm()* returnes if currently running image is confirmed.

```
int nxboot_get_confirm(void);
```

Confirm performed with *nxboot_confirm()* function. The call is idempotent.

```
int nxboot_confirm(void);
```

Application access (ii)

Gets more descriptive state of bootlader and its next operation on reboot.

```
struct nxboot_state
{
  int update /* Number of update slot */
  int recovery; /* Number of recovery slot */
  bool recovery_valid; /* True if recovery image is valid */
  bool recovery_present; /* True if the image in primary has a recovery */
  bool primary_confirmed; /* True if primary slot is confirmed */
  enum nxboot_update_type next_boot;
};

int nxboot_get_state(struct nxboot_state *state);
```

Application access (iii)

```c
int nxboot_ramcopy(void);
```

- copies bootable image from primary partition to RAM

```
CONFIG_NXBOOT_COPY_TO_RAM=y
CONFIG_NXBOOT_RAMSTART=0x0
```

- credit for the patch goes to Tim Hardisty

3 Conclusion

Conclusion

- NuttX bootloader can provide reliable and fast firmware update with possible backup
- the algorithm is considerate to flash wear
- almost ad lib tool can be used to upload firmware (simple TCP server-client, serial port, sd card, ...)
- integrated in NuttX apps directory (*boot/nxboot*)
- TODOS:
  - cryptography?
  - requires more space in memory, two slot alternative for devices without external NOR

References

- Documentation: https://nuttx.apache.org/docs/latest/applications/boot/nxboot/index.html