Boot Loader

Summary

mcuboot comprises two packages:

- The bootutil library (boot/bootutil)
- The boot application (each port has its own at boot/)

The bootutil library performs most of the functions of a boot loader. In particular, the piece that is missing is the final step of actually jumping to the main image. This last step is instead implemented by the boot application. Boot loader functionality is separated in this manner to enable unit testing of the boot loader. A library can be unit tested, but an application can't. Therefore, functionality is delegated to the bootutil library when possible.

Limitations

The boot loader currently only supports images with the following characteristics:

- Built to run from flash.
- Built to run from a fixed location (i.e., not position-independent).

Image Format

The following definitions describe the image format.

```
#define IMAGE_MAGIC                 0x96f3b83d

#define IMAGE_HEADER_SIZE           32

struct image_version {
    uint8_t iv_major;
    uint8_t iv_minor;
    uint16_t iv_revision;
    uint32_t iv_build_num;
};

/** Image header.  All fields are in little endian byte order. */
struct image_header {
    uint32_t ih_magic;
    uint32_t ih_load_addr;
    uint16_t ih_hdr_size;           /* Size of image header
(bytes). */
    uint16_t ih_protect_tlv_size;   /* Size of protected TLV area
(bytes). */
    uint32_t ih_img_size;           /* Does not include header. */
    uint32_t ih_flags;              /* IMAGE_F_[...]. */
    struct image_version ih_ver;
    uint32_t _pad1;
};

#define IMAGE_TLV_INFO_MAGIC        0x6907
#define IMAGE_TLV_PROT_INFO_MAGIC   0x6908

/** Image TLV header.  All fields in little endian. */
struct image_tlv_info {
    uint16_t it_magic;
    uint16_t it_tlv_tot;  /* size of TLV area (including tlv_info
header) */
};

/** Image trailer TLV format. All fields in little endian. */
struct image_tlv {
    uint8_t  it_type;   /* IMAGE_TLV_[...]. */
    uint8_t  _pad;
    uint16_t it_len;    /* Data length (not including TLV header).
*/
};

/*
 * Image header flags.
 */
#define IMAGE_F_PIC                      0x00000001 /* Not
supported. */
```

```
#define IMAGE_F_NON_BOOTABLE             0x00000010 /* Split image
app. */
#define IMAGE_F_RAM_LOAD                 0x00000020

/*
 * Image trailer TLV types.
 */
#define IMAGE_TLV_KEYHASH           0x01   /* hash of the public
key */
#define IMAGE_TLV_SHA256            0x10   /* SHA256 of image hdr
and body */
#define IMAGE_TLV_RSA2048_PSS       0x20   /* RSA2048 of hash
output */
#define IMAGE_TLV_ECDSA224          0x21   /* ECDSA of hash output
*/
#define IMAGE_TLV_ECDSA256          0x22   /* ECDSA of hash output
*/
#define IMAGE_TLV_RSA3072_PSS       0x23   /* RSA3072 of hash
output */
#define IMAGE_TLV_ED25519           0x24   /* ED25519 of hash
output */
#define IMAGE_TLV_ENC_RSA2048       0x30   /* Key encrypted with
RSA-OAEP-2048 */
#define IMAGE_TLV_ENC_KW128         0x31   /* Key encrypted with
AES-KW-128 */
#define IMAGE_TLV_ENC_EC256         0x32   /* Key encrypted with
ECIES P256 */
#define IMAGE_TLV_DEPENDENCY        0x40   /* Image depends on
other image */
#define IMAGE_TLV_SEC_CNT           0x50   /* security counter */
```

Optional type-length-value records (TLVs) containing image metadata are placed after the end of the image.

The `ih_protect_tlv_size` field indicates the length of the protected TLV area. If protected TLVs are present then a TLV info header with magic equal to `IMAGE_TLV_PROT_INFO_MAGIC` must be present and the protected TLVs (plus the info header itself) have to be included in the hash calculation. Otherwise the hash is only calculated over the image header and the image itself. In this case the value of the `ih_protect_tlv_size` field is 0.

The `ih_hdr_size` field indicates the length of the header, and therefore the offset of the image itself. This field provides for backwards compatibility in case of changes to the format of the image header.

Flash Map

A device's flash is partitioned according to its *flash map*. At a high level, the flash map maps numeric IDs to *flash areas*. A flash area is a region of disk with the following properties:

1. An area can be fully erased without affecting any other areas.
2. A write to one area does not restrict writes to other areas.

The boot loader uses the following flash area IDs:

```
/* Independent from multiple image boot */
#define FLASH_AREA_BOOTLOADER         0
#define FLASH_AREA_IMAGE_SCRATCH      3
```

```
/* If the boot loader is working with the first image */
#define FLASH_AREA_IMAGE_PRIMARY      1
#define FLASH_AREA_IMAGE_SECONDARY    2
```

```
/* If the boot loader is working with the second image */
#define FLASH_AREA_IMAGE_PRIMARY      5
#define FLASH_AREA_IMAGE_SECONDARY    6
```

The bootloader area contains the bootloader image itself. The other areas are described in subsequent sections. The flash could contain multiple executable images therefore the flash area IDs of primary and secondary areas are mapped based on the number of the active image (on which the bootloader is currently working).

Image Slots

A portion of the flash memory can be partitioned into multiple image areas, each contains two image slots: a primary slot and a secondary slot.
The boot loader will only run an image from the primary slot, so images must be built such that they can run from that fixed location in flash. If the boot loader needs to run the image resident in the secondary slot, it must copy its contents into the primary slot before doing so, either by swapping the two images or by overwriting the contents of the primary slot. The bootloader supports either swap- or overwrite-based image upgrades, but must be configured at build time to choose one of these two strategies.

In addition to the slots of image areas, the boot loader requires a scratch area to allow for reliable image swapping. The scratch area must have a size that is enough to store at least the largest sector that is going to be swapped. Many devices have small equally sized flash sectors, eg 4K, while others have variable sized sectors where the largest sectors might be 128K or 256K, so the scratch must be big enough to store that. The scratch is only ever used when swapping firmware, which means only when doing an upgrade. Given that, the main reason for using a larger size for the scratch is that flash wear will be more evenly distributed, because a single sector would be written twice the number of times than using two sectors, for example. To evaluate the ideal size of the scratch for your use case the following parameters are relevant:

- the ratio of image size / scratch size
- the number of erase cycles supported by the flash hardware

The image size is used (instead of slot size) because only the slot's sectors that are actually used for storing the image are copied. The image/scratch ratio is the number of times the scratch will be erased on every upgrade. The number of erase cycles divided by the image/scratch ratio will give you the number of times an upgrade can be performed before the device goes out of spec.

```
num_upgrades = number_of_erase_cycles / (image_size / scratch_size)
```

Let's assume, for example, a device with 10000 erase cycles, an image size of 150K and a scratch of 4K (usual minimum size of 4K sector devices). This would result in a total of:

`10000 / (150 / 4) ~ 267`

Increasing the scratch to 16K would give us:

`10000 / (150 / 16) ~ 1067`

There is no *best* ratio, as the right size is use-case dependent. Factors to consider include the number of times a device will be upgraded both in the field and during development, as well as any desired safety margin on the manufacturer's specified number of erase cycles. In general, using a ratio that allows hundreds to thousands of field upgrades in production is recommended.

The overwrite upgrade strategy is substantially simpler to implement than the image swapping strategy, especially since the bootloader must work properly even when it is reset during the middle of an image swap. For this reason, the rest of the document describes its behavior when configured to swap images during an upgrade.

Boot Swap Types

When the device first boots under normal circumstances, there is an up-to-date firmware image in each primary slot, which mcuboot can validate and then chain-load. In this case, no image swaps are necessary. During device upgrades, however, new candidate image(s) is present in the secondary slot(s), which mcuboot must swap into the primary slot(s) before booting as discussed above.

Upgrading an old image with a new one by swapping can be a two-step process. In this process, mcuboot performs a "test" swap of image data in flash and boots the new image or it will be executed during operation. The new image can then update the contents of flash at runtime to mark itself "OK", and mcuboot will then still choose to run it during the next boot. When this happens, the swap is made "permanent". If this doesn't happen, mcuboot will perform a "revert" swap during the next boot by swapping the image(s) back into its original location(s) , and attempting to boot the old image(s).

Depending on the use case, the first swap can also be made permanent directly. In this case, mcuboot will never attempt to revert the images on the next reset.

Test swaps are supported to provide a rollback mechanism to prevent devices from becoming "bricked" by bad firmware. If the device crashes immediately upon booting a new (bad) image, mcuboot will revert to the old (working) image at the next device reset, rather than booting the bad image again. This allows device firmware to make test swaps permanent only after performing a self-test routine.

On startup, mcuboot inspects the contents of flash to decide for each images which of these "swap types" to perform; this decision determines how it proceeds.

The possible swap types, and their meanings, are:

- `BOOT_SWAP_TYPE_NONE`: The "usual" or "no upgrade" case; attempt to boot the contents of the primary slot.
- `BOOT_SWAP_TYPE_TEST`: Boot the contents of the secondary slot by swapping images. Unless the swap is made permanent, revert back on the next boot.
- `BOOT_SWAP_TYPE_PERM`: Permanently swap images, and boot the upgraded image
  firmware.
- `BOOT_SWAP_TYPE_REVERT`: A previous test swap was not made permanent; swap back to the old image whose data are now in the secondary slot. If the old image marks itself "OK" when it boots, the next boot will have swap type `BOOT_SWAP_TYPE_NONE`.
- `BOOT_SWAP_TYPE_FAIL`: Swap failed because image to be run is not valid.
- `BOOT_SWAP_TYPE_PANIC`: Swapping encountered an unrecoverable error.

The "swap type" is a high-level representation of the outcome of the boot. Subsequent sections describe how mcuboot determines the swap type from the bit-level contents of flash.

Image Trailer

For the bootloader to be able to determine the current state and what actions should be taken during the current boot operation, it uses metadata stored in the image flash areas. While swapping, some of this metadata is temporarily copied into and out of the scratch area.

This metadata is located at the end of the image flash areas, and is called an image trailer. An image trailer has the following structure:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    ~
~
    ~    Swap status (BOOT_MAX_IMG_SECTORS * min-write-size * 3)
~
    ~
~
```

```
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |                 Encryption key 0 (16 octets) [*]
|
    |
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |                 Encryption key 1 (16 octets) [*]
|
    |
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |                      Swap size (4 octets)
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |   Swap info   |           0xff padding (7 octets)
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |   Copy done   |           0xff padding (7 octets)
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |   Image OK    |           0xff padding (7 octets)
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |                       MAGIC (16 octets)
|
    |
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
```

[*]: Only present if the encryption option is enabled (`MCUBOOT_ENC_IMAGES`).

The offset immediately following such a record represents the start of the next flash area.

Note: "min-write-size" is a property of the flash hardware. If the hardware allows individual bytes to be written at arbitrary addresses, then min-write-size is 1. If the hardware only allows writes at even addresses, then min-write-size is 2, and so on.

An image trailer contains the following fields:

1. Swap status: A series of records which records the progress of an image swap. To swap entire images, data are swapped between the two image areas one or more sectors at a time, like this:

- sector data in the primary slot is copied into scratch, then erased
- sector data in the secondary slot is copied into the primary slot, then erased
- sector data in scratch is copied into the secondary slot

As it swaps images, the bootloader updates the swap status field in a way that allows it to compute how far this swap operation has progressed for each sector. The swap status field can thus used to resume a swap operation if the bootloader is halted while a swap operation is ongoing and later reset. The `BOOT_MAX_IMG_SECTORS` value is the configurable maximum number of sectors mcuboot supports for each image; its value defaults to 128, but allows for either decreasing this size, to limit RAM usage, or to increase it in devices that have massive amounts of Flash or very small sized sectors and thus require a bigger configuration to allow for the handling of all slot's sectors. The factor of min-write-sz is due to the behavior of flash hardware. The factor of 3 is explained below.

2. Encryption keys: key-encrypting keys (KEKs). These keys are needed for image encryption and decryption. See the encrypted images document for more information.
3. Swap size: When beginning a new swap operation, the total size that needs to be swapped (based on the slot with largest image + TLVs) is written to this location for easier recovery in case of a reset while performing the swap.
4. Swap info: A single byte which encodes the following information:
    - Swap type: Stored in bits 0-3. Indicating the type of swap operation in
      progress. When mcuboot resumes an interrupted swap, it uses this field to
      determine the type of operation to perform. This field contains one of the
      following values in the table below.
    - Image number: Stored in bits 4-7. It has always 0 value at single image
      boot. In case of multi image boot it indicates, which image was swapped when
      interrupt happened. The same scratch area is used during in case of all
      image swap operation. Therefore this field is used to determine which image
      the trailer belongs to if boot status is found on scratch area when the swap
      operation is resumed.

NAME	VALUE
`BOOT_SWAP_TYPE_TEST`	2
`BOOT_SWAP_TYPE_PERM`	3
`BOOT_SWAP_TYPE_REVERT`	4

5. Copy done: A single byte indicating whether the image in this slot is complete (0x01=done; 0xff=not done).

6. Image OK: A single byte indicating whether the image in this slot has been confirmed as good by the user (0x01=confirmed; 0xff=not confirmed).
7. MAGIC: The following 16 bytes, written in host-byte-order:

```
const uint32_t boot_img_magic[4] = {
    0xf395c277,
    0x7fefd260,
    0x0f505235,
    0x8079b62c,
};
```

IMAGE TRAILERS

At startup, the boot loader determines the boot swap type by inspecting the image trailers. When using the term "image trailers" what is meant is the aggregate information provided by both image slot's trailers.

New swaps (non-resumes)

For new swaps, mcuboot must inspect a collection of fields to determine which swap operation to perform.

The image trailers records are structured around the limitations imposed by flash hardware. As a consequence, they do not have a very intuitive design, and it is difficult to get a sense of the state of the device just by looking at the image trailers. It is better to map all the possible trailer states to the swap types described above via a set of tables. These tables are reproduced below.

Note: An important caveat about the tables described below is that they must be evaluated in the order presented here. Lower state numbers must have a higher priority when testing the image trailers.

```
State I
                 | primary slot | secondary slot |
-----------------+--------------+----------------|
           magic | Any          | Good           |
        image-ok | Any          | Unset          |
       copy-done | Any          | Any            |
-----------------+--------------+----------------'
 result: BOOT_SWAP_TYPE_TEST                     |
-------------------------------------------------'


State II
                 | primary slot | secondary slot |
-----------------+--------------+----------------|
           magic | Any          | Good           |
        image-ok | Any          | 0x01           |
       copy-done | Any          | Any            |
-----------------+--------------+----------------'
 result: BOOT_SWAP_TYPE_PERM                     |
-------------------------------------------------'
```

```
State III
                 | primary slot | secondary slot |
-----------------+--------------+----------------|
           magic | Good         | Unset          |
        image-ok | 0xff         | Any            |
       copy-done | 0x01         | Any            |
-----------------+--------------+----------------'
 result: BOOT_SWAP_TYPE_REVERT                   |
-------------------------------------------------'
```

Any of the above three states results in mcuboot attempting to swap images.

Otherwise, mcuboot does not attempt to swap images, resulting in one of the other three swap types, as illustrated by State IV.

```
State IV
                 | primary slot | secondary slot |
-----------------+--------------+----------------|
           magic | Any          | Any            |
        image-ok | Any          | Any            |
       copy-done | Any          | Any            |
-----------------+--------------+----------------'
 result: BOOT_SWAP_TYPE_NONE,                    |
         BOOT_SWAP_TYPE_FAIL, or                 |
         BOOT_SWAP_TYPE_PANIC                    |
-------------------------------------------------'
```

In State IV, when no errors occur, mcuboot will attempt to boot the contents of the primary slot directly, and the result is `BOOT_SWAP_TYPE_NONE`. If the image in the primary slot is not valid, the result is `BOOT_SWAP_TYPE_FAIL`. If a fatal error occurs during boot, the result is `BOOT_SWAP_TYPE_PANIC`. If the result is either `BOOT_SWAP_TYPE_FAIL` or `BOOT_SWAP_TYPE_PANIC`, mcuboot hangs rather than booting an invalid or compromised image.

Note: An important caveat to the above is the result when a swap is requested and the image in the secondary slot fails to validate, due to a hashing or signing error. This state behaves as State IV with the extra action of marking the image in the primary slot as "OK", to prevent further attempts to swap.

Resumed swaps

If mcuboot determines that it is resuming an interrupted swap (i.e., a reset occurred mid-swap), it fully determines the operation to resume by reading the `swap info` field from the active trailer and extracting the swap type from bits 0-3. The set of tables in the previous section are not necessary in the resume case.

High-Level Operation

With the terms defined, we can now explore the boot loader's operation. First, a high-level overview of the boot process is presented. Then, the following sections describe each step of the process in more detail.

Procedure:

1. Inspect swap status region; is an interrupted swap being resumed?
    - Yes: Complete the partial swap operation; skip to step 3.
    - No: Proceed to step 2.
2. Inspect image trailers; is a swap requested?
    - Yes:
        i. Is the requested image valid (integrity and security check)?
            - Yes.
              a. Perform swap operation.
              b. Persist completion of swap procedure to image trailers.
              c. Proceed to step 3.
            - No.
              a. Erase invalid image.
              b. Persist failure of swap procedure to image trailers.
              c. Proceed to step 3.
    - No: Proceed to step 3.
3. Boot into image in primary slot.

Multiple Image Boot

When the flash contains multiple executable images the boot loader's operation is a bit more complex but similar to the previously described procedure with one image. Every image can be updated independently therefore the flash is partitioned further to arrange two slots for each image.

```
+--------------------+
| MCUBoot            |
+--------------------+
        ~~~~~            <- memory might be not contiguous
+--------------------+
| Image 0            |
| primary   slot     |
+--------------------+
| Image 0            |
| secondary slot     |
+--------------------+
        ~~~~~            <- memory might be not contiguous
+--------------------+
| Image N            |
| primary   slot     |
+--------------------+
| Image N            |
```

```
| secondary slot     |
+--------------------+
| Scratch            |
+--------------------+
```

MCUBoot is also capable of handling dependencies between images. For example
if an image needs to be reverted it might be necessary to revert another one too
(e.g. due to API incompatibilities) or simply to prevent from being updated
because of an unsatisfied dependency. Therefore all aborted swaps have to be
completed and all the swap types have to be determined for each image before
the dependency checks. Dependency handling is described in more detail in a
following section. The multiple image boot procedure is organized in loops which
iterate over all the firmware images. The high-level overview of the boot
process is presented below.

- **Loop 1. Iterate over all images**

  a. Inspect swap status region of current image; is an interrupted swap being
  resumed?

     - Yes:

       - Review the validity of previously determined swap types
         of other images.
       - Complete the partial swap operation.
       - Mark the swap type as `None`.
       - Skip to next image.
     - No: Proceed to step 2.

  b. Inspect image trailers in the primary and secondary slot; is an image
  swap requested?

     - Yes: Review the validity of previously determined swap types of other
       images. Is the requested image valid (integrity and security
       check)?

       - Yes:

         - Set the previously determined swap type for the current image.
         - Skip to next image.
       - No:

         - Erase invalid image.
         - Persist failure of swap procedure to image trailers.
         - Mark the swap type as `Fail`.
         - Skip to next image.
     - No:

       - Mark the swap type as `None`.
       - Skip to next image.

- **Loop 2. Iterate over all images**

    a. Does the current image depend on other image(s)?

    - Yes: Are all the image dependencies satisfied?
        - Yes: Skip to next image.
        - No:
            - Modify swap type depending on what the previous type was.
            - Restart dependency check from the first image.
    - No: Skip to next image.
- **Loop 3. Iterate over all images**

    a. Is an image swap requested?

    - Yes:
        - Perform image update operation.
        - Persist completion of swap procedure to image trailers.
        - Skip to next image.
    - No: Skip to next image.
- **Loop 4. Iterate over all images**

    a. Validate image in the primary slot (integrity and security check) or at least do a basic sanity check to avoid booting into an empty flash area.
- Boot into image in the primary slot of the 0th image position\
(other image in the boot chain is started by another image).

Image Swapping

The boot loader swaps the contents of the two image slots for two reasons:

- User has issued a "set pending" operation; the image in the secondary slot should be run once (state I) or repeatedly (state II), depending on whether a permanent swap was specified.
- Test image rebooted without being confirmed; the boot loader should revert to the original image currently in the secondary slot (state III).

If the image trailers indicates that the image in the secondary slot should be run, the boot loader needs to copy it to the primary slot. The image currently in the primary slot also needs to be retained in flash so that it can be used later. Furthermore, both images need to be recoverable if the boot loader resets in the middle of the swap operation. The two images are swapped according to the following procedure:

1. Determine if both slots are compatible enough to have their images swapped. To be compatible, both have to have only sectors that can fit into the scratch area and if one of them has larger sectors than the other, it must be able to entirely fit some rounded number of sectors from the other slot. In the next steps we'll use the terminology "region" for the total amount of

data copied/erased because this can be any amount of sectors depending on
how many the scratch is able to fit for some swap operation.

2. Iterate the list of region indices in descending order (i.e., starting
with the greatest index); only regions that are predetermined to be part of
the image are copied; current element = "index".

    - a. Erase scratch area.
    - b. Copy secondary_slot[index] to scratch area.
        - If this is the last region in the slot, scratch area has a temporary
status area initialized to store the initial state, because the primary slot's last region will have to be erased. In this case,
only the data that was calculated to amount to the image is copied.
        - Else if this is the first swapped region but not the last region in
the slot, initialize the status area in primary slot and copy the
full region contents.
        - Else, copy entire region contents.
    - c. Write updated swap status (i).
    - d. Erase secondary_slot[index]
    - e. Copy primary_slot[index] to secondary_slot[index] according to amount
previously copied at step b.
        - If this is not the last region in the slot, erase the trailer in the
secondary slot, to always use the one in the primary slot.
    - f. Write updated swap status (ii).
    - g. Erase primary_slot[index].
    - h. Copy scratch area to primary_slot[index] according to amount previously copied at step b.
        - If this is the last region in the slot, the status is read from scratch (where it was stored temporarily) and written anew in the
primary slot.
    - i. Write updated swap status (iii).

3. Persist completion of swap procedure to the primary slot image trailer.

The additional caveats in step 2f are necessary so that the secondary slot image
trailer can be written by the user at a later time. With the image trailer
unwritten, the user can test the image in the secondary slot
(i.e., transition to state I).

Note1: If the region being copied contains the last sector, then swap status is
temporarily maintained on scratch for the duration of this operation, always
using the primary slot's area otherwise.

Note2: The bootloader tries to copy only used sectors (based on largest image
installed on any of the slots), minimizing the amount of sectors copied and
reducing the amount of time required for a swap operation.

The particulars of step 3 vary depending on whether an image is being tested, permanently used, reverted or a validation failure of the secondary slot happened when a swap was requested:

```
* test:
    o Write primary_slot.copy_done = 1
    (swap caused the following values to be written:
        primary_slot.magic = BOOT_MAGIC
        secondary_slot.magic = UNSET
        primary_slot.image_ok = Unset)

* permanent:
    o Write primary_slot.copy_done = 1
    (swap caused the following values to be written:
        primary_slot.magic = BOOT_MAGIC
        secondary_slot.magic = UNSET
        primary_slot.image_ok = 0x01)

* revert:
    o Write primary_slot.copy_done = 1
    o Write primary_slot.image_ok = 1
    (swap caused the following values to be written:
        primary_slot.magic = BOOT_MAGIC)

* failure to validate the secondary slot:
    o Write primary_slot.image_ok = 1
```

After completing the operations as described above the image in the primary slot should be booted.

Swap Status

The swap status region allows the boot loader to recover in case it restarts in the middle of an image swap operation. The swap status region consists of a series of single-byte records. These records are written independently, and therefore must be padded according to the minimum write size imposed by the flash hardware. In the below figure, a min-write-size of 1 is assumed for simplicity. The structure of the swap status region is illustrated below. In this figure, a min-write-size of 1 is assumed for simplicity.

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |sec127,state 0 |sec127,state 1 |sec127,state 2 |sec126,state 0
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |sec126,state 1 |sec126,state 2 |sec125,state 0 |sec125,state 1
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    |sec125,state 2 |
|
    +-+-+-+-+-+-+-+-+
+
    ~
~
    ~               [Records for indices 124 through 1
~
    ~
~
    ~               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
    ~               |sec000,state 0 |sec000,state 1 |sec000,state 2
|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
```

The above is probably not helpful at all; here is a description in English.

Each image slot is partitioned into a sequence of flash sectors. If we were to enumerate the sectors in a single slot, starting at 0, we would have a list of sector indices. Since there are two image slots, each sector index would correspond to a pair of sectors. For example, sector index 0 corresponds to the first sector in the primary slot and the first sector in the secondary slot. Finally, reverse the list of indices such that the list starts with index `BOOT_MAX_IMG_SECTORS - 1` and ends with 0. The swap status region is a representation of this reversed list.

During a swap operation, each sector index transitions through four separate states:

```
0. primary slot: image 0,   secondary slot: image 1,   scratch: N/A
1. primary slot: image 0,   secondary slot: N/A,       scratch:
image 1 (1->s, erase 1)
2. primary slot: N/A,       secondary slot: image 0,   scratch:
image 1 (0->1, erase 0)
3. primary slot: image 1,   secondary slot: image 0,   scratch: N/A
    (s->0)
```

Each time a sector index transitions to a new state, the boot loader writes a record to the swap status region. Logically, the boot loader only needs one record per sector index to keep track of the current swap state. However, due to limitations imposed by flash hardware, a record cannot be overwritten when an index's state changes. To solve this problem, the boot loader uses three records per sector index rather than just one.

Each sector-state pair is represented as a set of three records. The record values map to the above four states as follows

```
          | rec0 | rec1 | rec2
  --------+------+------+------
  state 0 | 0xff | 0xff | 0xff
  state 1 | 0x01 | 0xff | 0xff
  state 2 | 0x01 | 0x02 | 0xff
  state 3 | 0x01 | 0x02 | 0x03
```

The swap status region can accommodate `BOOT_MAX_IMG_SECTORS` sector indices. Hence, the size of the region, in bytes, is `BOOT_MAX_IMG_SECTORS * min-write-size * 3`. The only requirement for the index count is that it is great enough to account for a maximum-sized image (i.e., at least as great as the total sector count in an image slot). If a device's image slots have been configured with `BOOT_MAX_IMG_SECTORS: 128` and use less than 128 sectors, the first record that gets written will be somewhere in the middle of the region. For example, if a slot uses 64 sectors, the first sector index that gets swapped is 63, which corresponds to the exact halfway point within the region.

Note: since the scratch area only ever needs to record swapping of the last sector, it uses at most min-write-size * 3 bytes for its own status area.

Reset Recovery

If the boot loader resets in the middle of a swap operation, the two images may be discontiguous in flash. Bootutil recovers from this condition by using the image trailers to determine how the image parts are distributed in flash.

The first step is determine where the relevant swap status region is located. Because this region is embedded within the image slots, its location in flash changes during a swap operation. The below set of tables map image trailers contents to swap status location. In these tables, the "source" field indicates where the swap status region is located. In case of multi image boot the images primary area and the single scratch area is always examined in pairs. If swap status found on scratch area then it might not belong to the current image. The swap_info field of swap status stores the corresponding image number. If it does not match then "source: none" is returned.

```
            | primary slot | scratch      |
  ----------+--------------+--------------|
      magic | Good         | Any          |
  copy-done | 0x01         | N/A          |
  ----------+--------------+--------------'
  source: none                            |
  ----------------------------------------'

            | primary slot | scratch      |
  ----------+--------------+--------------|
      magic | Good         | Any          |
  copy-done | 0xff         | N/A          |
  ----------+--------------+--------------'
  source: primary slot                    |
  ----------------------------------------'
```

```
              | primary slot | scratch      |
    ----------+--------------+--------------|
        magic | Any          | Good         |
    copy-done | Any          | N/A          |
    ----------+--------------+--------------'
    source: scratch                         |
    ----------------------------------------'

              | primary slot | scratch      |
    ----------+--------------+--------------|
        magic | Unset        | Any          |
    copy-done | 0xff         | N/A          |
    ----------+--------------+--------------|
    source: primary slot                    |
    ----------------------------------------+----------------------
--------+
    This represents one of two cases:
       |
    o No swaps ever (no status to read, so no harm in checking).
        |
    o Mid-revert; status in the primary slot.
       |
    For this reason we assume the primary slot as source, to
trigger a      |
    check of the status area and find out if there was swapping
under way. |
    ---------------------------------------------------------------
--------'
```

If the swap status region indicates that the images are not contiguous, mcuboot determines the type of swap operation that was interrupted by reading the `swap info` field in the active image trailer and extracting the swap type from bits 0-3 then resumes the operation. In other words, it applies the procedure defined in the previous section, moving image 1 into the primary slot and image 0 into the secondary slot. If the boot status indicates that an image part is present in the scratch area, this part is copied into the correct location by starting at step e or step h in the area-swap procedure, depending on whether the part belongs to image 0 or image 1.

After the swap operation has been completed, the boot loader proceeds as though it had just been started.

Integrity Check

An image is checked for integrity immediately before it gets copied into the primary slot. If the boot loader doesn't perform an image swap, then it can perform an optional integrity check of the image in the primary slot if `MCUBOOT_VALIDATE_PRIMARY_SLOT` is set, otherwise it doesn't perform an integrity check.

During the integrity check, the boot loader verifies the following aspects of an image:

- 32-bit magic number must be correct (`IMAGE_MAGIC`).
- Image must contain an `image_tlv_info` struct, identified by its magic (`IMAGE_TLV_PROT_INFO_MAGIC` or `IMAGE_TLV_INFO_MAGIC`) exactly following the firmware (`hdr_size` + `img_size`). If `IMAGE_TLV_PROT_INFO_MAGIC` is found then after `ih_protect_tlv_size` bytes, another `image_tlv_info` with magic equal to `IMAGE_TLV_INFO_MAGIC` must be present.
- Image must contain a SHA256 TLV.
- Calculated SHA256 must match SHA256 TLV contents.
- Image *may* contain a signature TLV. If it does, it must also have a KEYHASH TLV with the hash of the key that was used to sign. The list of keys will then be iterated over looking for the matching key, which then will then be used to verify the image contents.

Security

As indicated above, the final step of the integrity check is signature verification. The boot loader can have one or more public keys embedded in it at build time. During signature verification, the boot loader verifies that an image was signed with a private key that corresponds to the embedded KEYHASH TLV.

For information on embedding public keys in the boot loader, as well as producing signed images, see: signed_images.

If you want to enable and use encrypted images, see: encrypted_images.

Protected TLVs

If the TLV area contains protected TLV entries, by beginning with a `struct image_tlv_info` with a magic value of `IMAGE_TLV_PROT_INFO_MAGIC` then the data of those TLVs must also be integrity and authenticity protected. Beyond the full size of the protected TLVs being stored in the `image_tlv_info`, the size of the protected TLVs together with the size of the `image_tlv_info` struct itself are also saved in the `ih_protected_size` field inside the header.

Whenever an image has protected TLVs the SHA256 has to be calculated over not just the image header and the image but also the TLV info header and the protected TLVs.

```
A +---------------------+
  | Header              | <- struct image_header
  +---------------------+
  | Payload             |
  +---------------------+
  | TLV area            |
  | +-----------------+ |    struct image_tlv_info with
  | | TLV area header | | <- IMAGE_TLV_PROT_INFO_MAGIC (optional)
  | +------------------+ |
  | | Protected TLVs   | | <- Protected TLVs (struct image_tlv)
B | +------------------+ |
  | | TLV area header  | | <- struct image_tlv_info with
IMAGE_TLV_INFO_MAGIC
C | +------------------+ |
  | | SHA256 hash      | | <- hash from A - B (struct image_tlv)
D | +------------------+ |
  | | Keyhash          | | <- indicates which pub. key for sig
(struct image_tlv)
  | +------------------+ |
  | | Signature        | | <- signature from C - D (struct
image_tlv), only hash
  | +------------------+ |
  +----------------------+
```

Dependency Check

MCUBoot can handle multiple firmware images. It is possible to update them independently but in many cases it can be desired to be able to describe dependencies between the images (e.g. to ensure API compliance and avoid interoperability issues).

The dependencies between images can be described with additional TLV entries in the protected TLV area after the end of an image. There can be more than one dependency entry, but in practice if the platform only supports two individual images then there can be maximum one entry which reflects to the other image.

At the phase of dependency check all aborted swaps are finalized if there were any. During the dependency check the boot loader verifies whether the image dependencies are all satisfied. If at least one of the dependencies of an image is not fulfilled then the swap type of that image has to be modified accordingly and the dependency check needs to be restarted. This way the number of unsatisfied dependencies will decrease or remain the same. There is always at least 1 valid configuration. In worst case, the system returns to the initial state after dependency check.

For more information on adding dependency entries to an image, see: imgtool.

Downgrade Prevention

Downgrade prevention is a feature which enforces that the new image must have a higher version/security counter number than the image it is replacing, thus preventing the malicious downgrading of the device to an older and possibly vulnerable version of its firmware.

SW Based Downgrade Prevention

During the software based downgrade prevention the image version numbers are compared. This feature is enabled with the `MCUBOOT_DOWNGRADE_PREVENTION` option. In this case downgrade prevention is only available when the overwrite-based image update strategy is used (i.e. `MCUBOOT_OVERWRITE_ONLY` is set).

HW Based Downgrade Prevention

Each signed image can contain a security counter in its protected TLV area. During the hardware based downgrade prevention (alias rollback protection) the new image's security counter will be compared with the currently active security counter value which must be stored in a non-volatile and trusted component of the device. This feature is enabled with the `MCUBOOT_HW_ROLLBACK_PROT` option. It is beneficial to handle this counter independently from image version number:

- It does not need to increase with each software release,
- It makes it possible to do software downgrade to some extent: if the security counter has the same value in the older image then it is accepted.