libFirmware Documentation

Release 0.8.0

ANSSI

Aug 26, 2019

CONTENTS

Contents

The firmware library is a toolkit for manipulating firmware-related informations and data. Its goal is to handle two separated usage:

1. Handling run state (detecting firmware mode and bank being executed)
2. Manipulating firmware files and associated metainformations (header, CRC. . .)

CHAPTER

ONE

OVERVIEW

1.1 Run mode usage

When using the libfirmware as a run state detection backend, it can be used by any application to detect in which run mode and bank it is being executed. This allows executable that are compiled for multiple banks and execution mode to react differently depending on the current bank and mode.

The libfirmware supports the following, device-generic banks:

- Flip bank
- Flop bank

These banks are generic, dual-bank based resilient firmware images.

The libfirmware supports the following, device-generic run mode:

- FW mode (aka. nominal mode)
- DFU mode (aka. upgrade mode)

We consider these two run mode also as generic modes.

The libfirmware is based on ldscript variables to detect bank and mode. As a consequence, it is up to the linker to define the following variables:

```
__is_flip
__is_flop
__is_fw
__is_dfu
```

Each binary application should then have a *4-uplet* with specific addresses in its ldscript.

These variables are boolean variables, but as they are ldscript variables, only their addresses can be used. As a consequence, an address of 0xf0 (i.e. 240) is considered as *True*, any other address is considered as *False*.

These variables must be set at the begining of the ldscript file, out of any SECTIONS block. A typical definition would be:

```
__is_flip = 240;
__is_flop = 0;
__is_fw   = 240;
__is_dfu  = 0;
```

This definition define the application as being executed in flip mode, in nominal (i.e. FW) mode.

Hint: It is possible to use the libfirmware without differenciate run mode or with a single bank. As variables are boolean, they can be always defined as false

1.2 Firmware image manipulation

Manipulating firmware images should be the job of a dedicated task. This task **must** hold the TSK_UPGRADE permission to use this part of the libfirmware.

Hint: Tasks that only manipulate run mode will have this part of the libfirmware removed from the generated binary at link time, as these functions are not called

Manipulating firmware is more complex than only handling raw binary file. In Wookey (and as a consequence libfirmware) firmware images host header informations which contains various data, including cryptographic signatures, version, and various meta-data.

The libfirmware provides helper functions to manipulate such header, including parsing, version checking and so on.

The libfirmware also handle the firmware storage backend, including the firmware image update and the firmware bootloader metainformation header update.

Warning: All the header related API is specific to the way firmware headers and bootloader header update are handled. This part of the libfirmware can be ported from one hardware to another, but is sticked to the way Wookey is handling its upgrade

Manipulating the firmware storage backend requires to know precisely how the storage is structured. The libfirmware needs various informations:

- Each bank base address
- Each bootinfo base address
- The size of a bank

There is no specific constraints on how the banks, the bootloader(s) and the bootinfo headers are located in the storage area, although:

1. Each bank must have its own bootinfo
2. Each bootinfo must be stored in a dedicated flash sector, as the sector is fully erased at haeder update time and the checksum is calculated on the overall sector size
3. Each bank must be contigous

There is no constraints on the contiguity between the bootinfo sector and the bank sector(s), as there is no constrain on the bootloader position, as the bootloader is never acceded.

Warning: Take a great care to properly separate each bank and each bank bootinfo information, to keep a correct resiliency between both banks

All the requested informations (bank base address, bootinfo base address and bank size) is configured through the Kconfig mechanism of the libfirmware.

Caution: The libfirmware configuration must be done through the overall project configuration system (e.g. using the tataouine SDK) in order to correctly generate autoconf.h header file

CHAPTER

TWO

API

2.1 The Run mode API

Detecting current run mode can be done easily using the following API:

```
#include "libfw.h"

bool is_in_flip_mode(void);
bool is_in_flop_mode(void);

bool is_in_fw_mode(void);
bool is_in_dfu_mode(void);
```

These functions handle the ldscripts variable and return the correct state to the task.

These four functions should be enough for any task source that can be executed in various mode and state.

Hint: The executable can be compiled in PIE mode, as the differenciation is made at link time

2.2 Firmware image manipulation

Caution: These functions require the caller to hold the TSK_UPGRADE permission to work properly

Danger: The permission check is hold by the firmware storage driver (here the flash driver). When using another firmware storage driver, take a great care to check that the upgrade permission is set before mapping the device

The firmware image manipulation is composed of the following sets:

- firmware header manipulation
- firmware storage backend access
- bootloader bootinfo header upgrade

2.2.1 Manipulating firmware header

The firmware header is created at the begining of the firmware file, and is checked by the security monitor. In Wookey, this task is made in association with the Secure Element in order to handle cryptographic content of the header.

Although, as the header comes from an external host through a USB layer, the header has to be translated in the device endianess, and converted from a raw buffer to a real effective header.

This is done using the following API:

```
#include "libfw.h"

int firmware_parse_header(__in  const uint8_t      *buffer,
                          __in  const uint32_t      len,
                          __in  const uint32_t      siglen,
                          __out firmware_header_t *header,
                          __out uint8_t           *sig);
```

__out parameters are updated by the firmware_parse_header() function, based on the input buffer given. The signature is extracted from the input buffer in order to be used in future signature check of the header.

> **Warning:** parameters must be allocated content, as this function doesn't allocate anything

Translating an existing formated header into raw data can also be done using the following API:

```
#include "libfw.h"

int firmware_header_to_raw(__in const firmware_header_t *header,
                           __out  uint8_t     *buffer,
                           __out  const uint32_t     len);
```

When the header parsing fails, it is possible to dump on the serial line the haeder content, using the following API:

```
#include "libfw.h"

void firmware_print_header(const firmware_header_t * header);
```

> **Warning:** This function is useless in production mode, as the serial line is deactivated. This function is for debug purpose only

By now, firmware files are bank-specific (i.e. a firmware file is specific to flip or to flop). This restriction is due to various slotting constraints.

As a consequence, the libfirmware provice high level API to check the currently received firmware destination bank:

```
#include "libfw.h"

bool firmware_is_partition_flip(__in const firmware_header_t *header);
bool firmware_is_partition_flop(__in const firmware_header_t *header);
```

Hint: An invalid bank (i.e. FLIP while in flip mode or FLOP while in flop mode) should results in refusing the firmware. In Wookey, the DFU handling allows to refuse an authenticated but invalid bank firmware and to rollback for another firmware reception without rebooting the device

---

2.2.2 Accessing firmware storage backend

After having received the firmware header and validated that the header is authenticated, the firmware content is received chunk after chunk.

From now on, we have to store each chunk in the storage backend, starting with the target bank starting point.

Before that, we have to map the storage backend correctly. This is done by the initialization functions:

```
#include "libfw.h"

uint8_t firmware_early_init(t_device_mapping *devmap);

uint8_t firmware_init(void);
```

The firmware_early_init() function must be called during the initialization process, as it request a hardware ressource (the storage backend device).

This function requires as first parameter a devmap. This devmap is declared by the flash driver API and describes which part of the flash should be mapped.

The flash driver permits to map only a subset of the flash, based on the flip/flop structure.

A usual use of the devmap in this case would be to request the following map of the devmap structure:

- map_flip and map_flip_shr (**or** map_flop and map_flop_shr)
- map_ctrl

The map_flip (respectively map_flop) subdevice is the memory area containing the corresponding firmware.

The map_flip_shr (respectiverly map_flop_shr) subdevice is the memory area containing the bootloader corresponding bank boot header informations

Danger: It is useless (and dangerous) to request more. The initialization phase strict separation of the EwoK kernel avoid any further attempt to map other parts of the flash memory

As usual, the *firmware_init()* function initialize the flash device control structure.

Now that the flash device is ready, we can loop on the firmware chunk write action. This is done with the following API:

```
#include "libfw.h"

uint8_t fw_storage_prepare_access(void);
uint8_t fw_storage_write_buffer(physaddr_t dest, uint32_t *buffer, uint32_t size);
uint8_t fw_storage_finalize_access(void);
```

Danger: As flash subdevices are mapped in voluntary mode, use fw_storage_prepare_access() and fw_storage_finalize_access() to map/unmap the drvice from the memory layout of the task

Writing a buffer to the storage backend requires a destination address. The initial address, coresponding to the target bank base address, can be found using the following API:

```
#include "libfw.h"
```

(continues on next page)

(continued from previous page)

```
uint32_t firmware_get_flip_base_addr(void);
uint32_t firmware_get_flop_base_addr(void);
```

To avoid any overwrite attempt associated to a corrupted firmware file, the bank size can also be returned using the following API:

```
#include "libfw.h"

uint32_t firmware_get_flip_size(void);
uint32_t firmware_get_flop_size(void);
```

2.2.3 Updating bootinfo

When the firmware is fully written and its integrity has been checked in comparison with the signature received from the cryptographic header, the bootinfo of the corresponding bank can be updated.

The libfirmware handle the bootinfo header:

```
#include "libfw.h"

uint8_t set_fw_header(const firmware_header_t *dfu_header, const uint8_t *sig, const␣
↪uint8_t *hash);
```

This function generate a complete header structure at the begining of the header sector, which correspond to the address set in the USR_LIB_FIRMWARE_FL[IO]P_BOOTINFO_ADDR. To avoid any injection of content in the header sector, the *set_fw_header()* execute the following steps:

1. It erase the bootinfo sector*
2. It generate the haeder info in memory, and calculate a complete cheksum of the bootinfo sector, which will be fullfill with 0xff pattern after the header structure data. The CRC32 is calculated on the overall sector but the CRC32 field itself
3. It update the overall sector with the new content forged in memory

Any attempt to reboot before the header is fully written make the CRC32 calculation by the bootloader invalid.

The header also hold a SHA256 signature of the firmware bank, which will be checked by the bootloader at boot time to check the bank integrity at boot time

---

Hint: The cryptographic and checksum information written by the libfirmware permit to validate both the integrity of the bootinfo header and the associated firmware bank at each boot

---

2.2.4 Rollback protection

One of the basic attack on an upgradable device would be to load a previous, vulnerable, version of the firmware image in order to exploit a well-known vulnerability. The libfirmware provide an API to detect rollback attacks:

```
#include "libfw.h"

bool fw_is_rollback(firmware_header_t *header);
int  fw_version_compare(uint32_t version1, uint32_t version2);
```

These functions permit to compare the current firmware version (which is stored in the firmware header) with the current firmware version. *fw_is_rollback()* return true if the update is an effective rollback (i.e. current version is greater that the proposed one). *fw_version_compare()* return an integer with is less than, equal or greater than 0 if version1 is respectively older, equal or newer than version2.

CHAPTER

THREE

FAQ

3.1 Why voluntary updating bootinfo instead of adding them to the firmware file directly ?

As we are in small devices where the firmware file has to be written in place during its download, it is not possible to validate the firmware integrity and authenticity while the firmware is not fully written. As a consequence, a firmware image holding the bootinfo would be bootable even if the authentication or integrity check fails.