Flash driver Documentation

Release 0.8.0

ANSSI

Oct 10, 2019

TABLE OF CONTENTS

This library is an implementation of the STM32F4 flash device driver.

It provide an abstraction of the flash device interactions through high level API in order to read, write, (un)lock the flash memory.

The current implementation of this flash driver support various STM32 flash intgrated flash devices components, including 1M and 2M flash banks, with both signle and double banking.

CHAPTER

ONE

ABOUT THE FLASH DRIVER

1.1 Principles

1.1.1 About flash memory technology

A flash memory is made of *sectors* of various size. Each sector can be acceded independently and

Writing data to a flash memory sector can only be done by writing zeros (i.e. settings flash bits from 1 to 0). As a consequence, when writing data into a sector, the sector must be first reinitialized to a reset value, which correspond to resetting all the sector bits to 1. Once the sector reset is done, it is possible to set the requested bits to 0 to write the effective data in flash.

As a consequence, a usual flash write access is based on a successive sector reset and sector write sequence.

1.1.2 Flash banks and concurrent accesses

Some Flash devices (like the STM32F4 one) can support multi-bank. This mechanism permit to separate the flash memory in two independent memory banks of the same size, both composed of multiple sectors.

Separating the flash into multiple banks permit to support write access into a given bank without blocking a potential concurrent read access on the other.

Such behavior is requested when updating the flash content at run time.

1.1.3 Flash devices specific storage areas

Flash devices may support storage area with specific properties. On STM32F4 SoCs, the flash device support the following:

- OTP (One Time Programmable) memory area, which can be written only once induring the device lifecycle. Such memory area can be used, for example, to store certificate files
- System memory area, usually hosting the bootrom. This memory area is not writeable but is accessible through the flash device programmable interface.

CHAPTER

TWO

ABOUT THE FLASH DRIVER API

2.1 Initializing the flash driver

Initializing the flash driver is done with the following API:

```
#include "libflash.h"

int flash_device_early_init(t_device_mapping *devmap);
int flash_init(void);
```

The flash driver early initialization must be executed before the end of the task initialization phase (see EwoK kernel API). This initialization declare which of the various flash components are requested by the task.

In order to specify which subdevice need to be mapped, a dedicated structure, named t_device_mapping and composed of booleans, has to be passed to the flash driver initialization function.

Depending on this structure, the flash driver will request and map only the required flash components.

This components are:

- the flash memory bank1 (in flash dual bank mode)
- the flash memory bank2 (in flash dual bank mode)
- the flash global memory (in flash mono bank mode)
- the flash control registers (handling flash read and write access)
- the flash system memory
- the flash OTP (One Time Programmable) memory
- the flash OPT configuration registers for bank1 or the overall memory in mono bank mode (including, WDP and RDP fields
- the flash OPT configuration registers for bank2 (in dual bank mode only)

As all these memory areas are mapped in various, non-contigous area of the memory map, the flash driver handle the various flash components through a voluntary, on request, mapping.

2.2 Handling flash components mapping

Before manipulating any of the flash components that have been deeclared in the initialization phase, the task has to request its mapping voluntary.

Using EwoK, this is done using the following API:

```
#include "api/syscall.h"

e_syscall_ret sys_cfg(CFG_DEV_MAP, int *dev_desc);
e_syscall_ret sys_cfg(CFG_DEV_UNMAP, int *dev_desc);
```

Danger: The task handling the flash device **must** have the MAP_VOLUNTARY permission

The flash driver handling the device descriptors at initialization time. To get back the device descriptor from the flash driver, the following function must be called:

```
#include "libflash.h"

int flash_get_descriptor(t_flash_dev_id id);
```

the t_flash_dev_id is an enumerate which is structured like the t_device_map is. As you can see in libflash.h, each boolean of the t_device_map structure is associated to a specific t_flash_dev_id value, for example BANK1, BANK2, CTRL, OTP and so on.

Knowing that, it is possible to map the requested device using the following:

```
#include "api/syscall.h"
#include "libflash.h"

int my_dev_desc;
e_syscall_ret ret;

[...]

my_dev_desc = flash_get_descriptor(BANK1);
ret = sys_cfg(CFG_DEV_MAP, &my_dev_desc);
if (ret != SYS_E_DONE) {
   printf("Unable to map bank1 device");
   goto err;
}

/* manipulating bank 1 */
[...]

ret = sys_cfg(CFG_DEV_UNMAP, &my_dev_desc);
if (ret != SYS_E_DONE) {
   printf("Unable to unmap bank1 device");
   goto err;
}

return 0;
err:
return 1;
```

Danger: None of the flash driver functions handle the mapping/unmapping step. This step is under the responsability of the task and must be handled correctly.

2.3 (Un)locking flash memory

Flash memory is not writeable by default. In order to write in flash memory, a specific magic numbers must be set into the flash configuration register before any write attempt.

This unlocking action is handled by the *flash_lock()* function. Locking again the flash is done through *flash_unlock()*.

Warning: The flash_lock() and flash_unlock() functions require the CTRL flash device areas to be mapped when they are called

2.4 Accessing flash option registers

Option registers handle some security specific flash global properties such as RDP (Read access protection), WDP (Write access protection) and so on.

These registers are locked while a dedicated magic is not written in the flash option bytes. These options bytes are also mapped in the OPT_BANK1 and OPT_BANK2 memory areas.

In order to unlock these registers, the flash driver provides *flash_unlock_opt()* and *flash_lock_opt()* functions.

Warning: The flash_lock_opt() and flash_unlock_opt() functions require the OPT_BANK1 (and potentially OPT_BANK2) flash device areas to be mapped when they are called

2.5 Erasing flash data

The flash memory is composed of sector of various size.

The device support various erase mode:

- Erasing a sector
- Erasing a bank
- Erasing the entire flash

Erasing a sector is simplified by the flash driver. Based on a given physical address, the flash driver is able to return the associated sector identifier, that we can use to request a sector erase from the flash.

This is done using the following:

```
#include "libflash.h"

physaddr_t myaddr;
uint8_t  sector_id;

/* setting myaddr */
[...]

sector_id = flash_select_sector(myaddr);
if (sector_id == 255) {
    printf("address out of flash memory\n");
    return;
```

(continues on next page)

(continued from previous page)

```
}
flash_sector_erase(sector_id);
```

When handling dual bank flash device, erasing a full bank is done by successively select the bank and requesting a full erase.

This is done using the following:

```
#include "libflash.h"

flash_set_bank_conf(FLASH_BANK_1);
flash_mass_erase();
```

Danger: Beware when execute mass erase ! You may erase your own code if the erase mechanism is not correctly set !

Warning: Erasing the flash requires CTRL to be mapped

2.6 Writting data to flash

The flash device require various flash write mode. It is possible to write:

- bytes
- words (4 bytes)
- double words (8 bytes) (depending on the flash device input power mode)

The flash driver provides the following API to write into flash:

```
#include "libflash.h"

void flash_program_dword(uint64_t *addr, uint64_t value);
void flash_program_word(uint32_t *addr, uint32_t word);
void flash_program_byte(uint8_t *addr, uint8_t value);
```

Warning: Writing data to flash requires the corresponding bank area and CTRL to be mapped

2.7 Reading into flash

Reading into flash is a direct memory access into the flash bank area.

Although, for better readability, the flash driver provides the folllowing API:

```
#include "libflash.h"

void flash_read(uint8_t *buffer, physaddr_t addr, uint32_t size);
```

Warning: reading data from flash requires the corresponding bank area to be mapped

CHAPTER

THREE

FLASH DRIVER FAQ

3.1 Is the Flash driver handling 1M and 2M flash memory ?

Yes. The flash memory size is defined in Kconfig and has to be set at configuration time.

3.2 Is the flash driver support mono-bank and dual-bank mode ?

Yes. The flash memory bank mode is defined in Kconfig and has to be set at configuration time. All flash devices may not support both modes (for e.g. 2MB flash devices on STM32F4x9 only support dual bank mode.

3.3 Is there helper for RDP/WDP access ?

No. The OPT registers have to be set manually by now.

3.4 I can’t map all my devices ! Why ?

The MPU restriction may imply that you can’t map all the devices in the same time. You have to handle successive map/unmap actions to serialize your various flash components mapping instead of trying to map all of them in the same time.