Movidius
an Intel company

USB Firmware Update

Application Note

v1.0

Intel Movidius
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Table of Contents

1 Introduction

The MDK development board MV0182 or MV0212 provides an 8MB SPI Flash Memory that contains a bootloader application that is loaded and executed at power on by the Myriad processor.

This Flash memory can be written with an application using a debugger cable and issuing the command "`make flash MV_SOC_REV={Myriad_version}`", where {Myriad_version} can have the values: `ma2150`, `ma2155`,`ma2450`, `ma2455`. Before writing the flash it might be recommended to erase the flash memory, which can be done using the "`flash_erase`" make target instead of the "`flash`" make target.

2 Goals and Scenarios

In this application note we will introduce an example application that is provided with the Movidius Development Kit (MDK) to upload a new Firmware in the onboard flash and execute this application from the chip at boot.

In order to boot from the onboard Flash the DIPSW S1 Boot CTRL/Boot Config needs to be set as follows:

```
DIP - S1
1 2 3 4 5 6 7 8
0 0 0 0 1 1 0 0
0 = off
1 = on
```

The implementation integrates third party SW. And demonstrates how to use this third party SW to successfully download in the flash memory a new firmware.

3 USB Firmware Uploader application

The example provides two applications: (1) one that has a bootloader and needs to be flashed via a debugger (in case of a finished product this can be done usually at production stage) this application is the UsbLoaderApp, and (2) another application named symbolically to suggest that this application can be an image processing application, for example, called ImageProcApp that is compiled and packaged in a stripped down `.elf` file, that can be loaded via USB and executed at power on.

3.1 USB Loader Application

This application is responsible for:

- reading the `.elf` file from AP, and write it (over SPI) to the EEPROM (8MB FLASH) memory;
- checking if there is any application previously programmed in EEPROM, and if so, it should run it;
- checking whether the CRC of the code written into the Flash memory is identical with the code received

via USB, in order to detect potential Flash memory errors.

This application will have to be written in the Flash memory via a debugger connected to the MDK dev board. It will be written at the beginning of the Flash memory and should not be erased.

After the application is downloaded in the flash memory the MDK board should be power cycled in order to load execute it. If it has been loaded successfully, then the USB connection will recognize a new device.

When the Myriad 2 chip is powered, the boot code from ROM (0x76000000) will start to translate the .mvcmd and load the sections to DDR, and then run it (for the MV0182, or MV0212 make sure you have proper boot mode selected!)

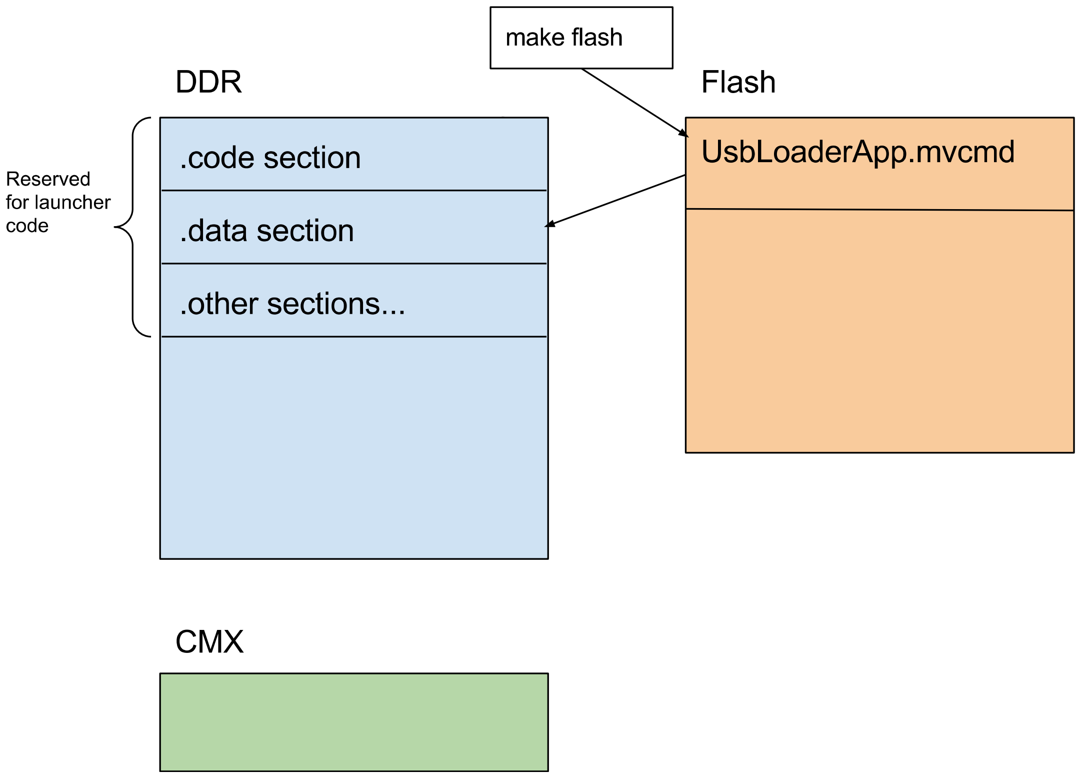
The data, code, stack and other sections of the UsbLoaderApp will be always configured to start from the beginning of DDR (text.start = 0x80000000). This part of DDR (reserved for the UsbLoaderApp section) will be treated by all user applications (ImageProcessingApp in this examples case) as RESERVED data, this is specified by using 2 defines:

```
#define DDR_RESERVER_FOR_LAUNCHER_START 0x80000000
```

and

```
#define DDR_RESERVED_FOR_LAUNCHER_END
( DDR_RESERVER_FOR_LAUNCHER_START + SIZEOF(<launcher application that
needs to be calculated>) + <4K flash sector alignment> )
```

NOTE: The user application (ImageProcessingApp in this example case) will include a common `.ldscript` where this reserved section is declared. *This is a very important note!* We can refer to this also as Application Launcher (which is actually the translated usbLoader.mvcmd from EEPROM).



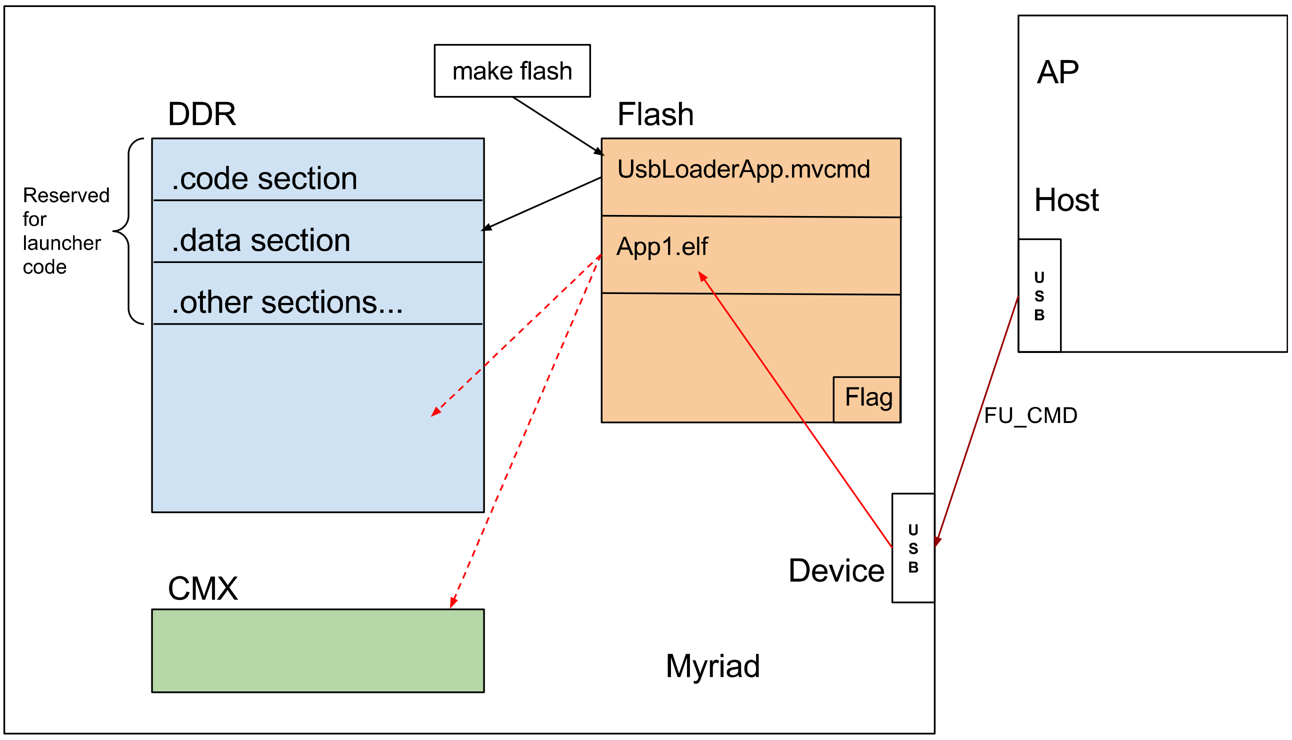
The UsbLoaderApp code will check (at the beginning of main function) if a global flag (this can be the last byte from EEPROM, or whatever predefined address in EEPROM) is:

- 0 (this means that no `.elf` is updated in the EEPROM), it will wait for AP to send an image (`.elf`) over USB
- 1 (this means we already have an `.elf` in the EEPROM) and we will execute it (details to be provided soon)

The UsbLoaderApp waits in a while loop for the `.elf` (if global flag is 0). Until a connection is established and a FU_CMD is received via USB. In case a firmware download is started it will load the received `.elf` file in a buffer and it will calculate the CRC on that buffer.

Once the entire `.elf` is received and the CRC is calculated it will write it into the FLASH memory, just below usbLoader.mvcmd. After successfully completing the writing operation it will re-read the data from the flash device and it will calculate the CRC of the data read from the FLASH memory and compare it with the one received.

If the written data is correct and the CRCs match it will call `swcLoadElf` in order to load the sections from `App1.elf`, and then it will call `DrvLeonOSStartup` (`&startOfApp1`) to start the App1 application.



This application can load and execute only one firmware. It cannot have more than one application loaded. However, in practice it would be possible to have more applications, but then each of them will need a different FLAG to signal their states, based on the states the bootloader can decide which of the apps to execute at power on.

The pseudocode for the implementation looks like in the example provided bellow:

```
{
  if(global_flag == 0)
  {
    // the code gets here only if there’s no application written to
EEPROM
    // or the AP asked for an firmware update
    waitElf(); //blocking till it gets and .elf file
    writeElfToEEPROM();
    goto runTheApp;
  }
  else if (global_flag == 1)
  {
    runTheApp:
    // This will load all needed sections for App1
    swcLoadElf(EEPROM_START+APPLICATION_OFFSET);
    // This is important so we avoid infinite loop at next board power
cycle
    setGlobalEEPROMFlag(1);
    DrvLeonOSStartup(&entryPointForApp1);
  }
}
```

3.2 Image Processing Application

In contrast with its name, this application only toggles one of the LEDs of the MDK development board (LED D11 will be blinking). However, in parallel to this led blinking task it also must configure a USB device and start the RTEMS thread that checks for USB firmware upload commands, in which case it will set the FLAG in flash memory to 0 and start the UsbLoaderApp from the FLASH memory.

The pseudocode for this handler should look like this:

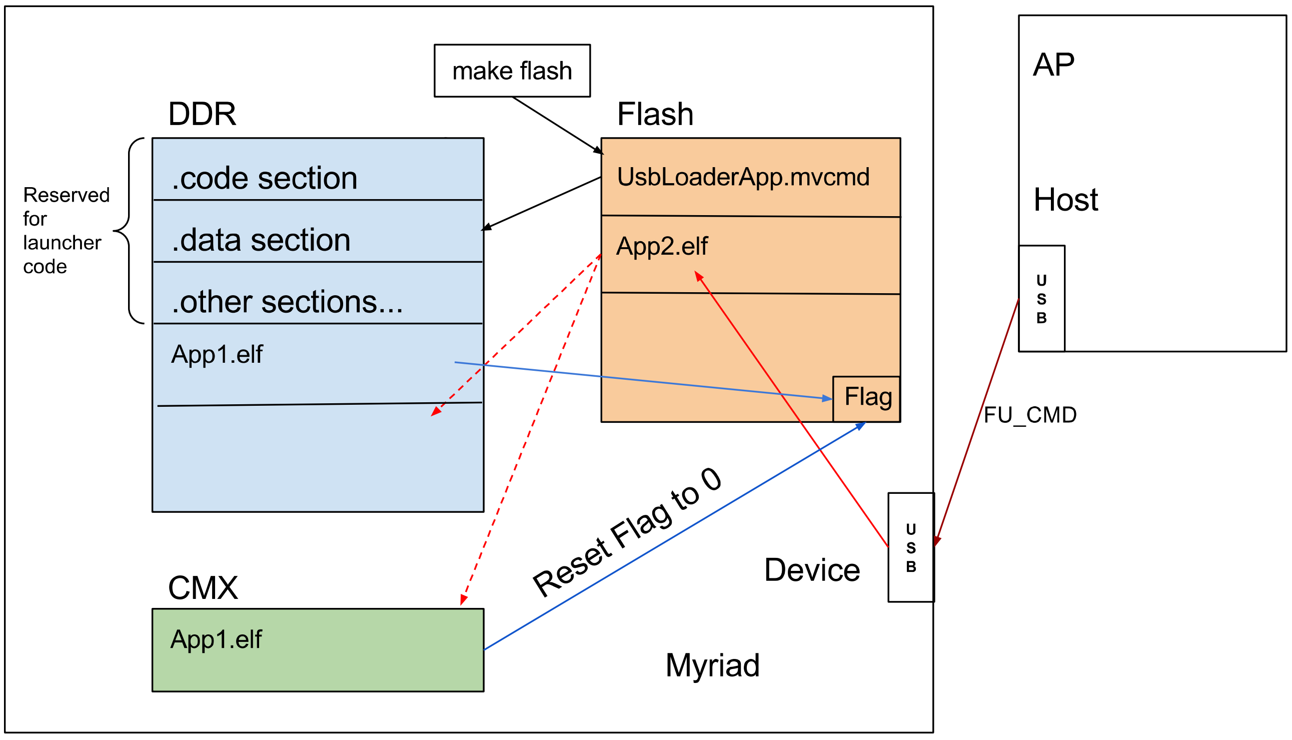
```
void FU_CMDHandler()
{
  // This will make the usbLoader wait for the .elf from AP;
  setGlobalEEPROMFlag(0);
  // Go to usbLoader code so you can receive the new elf
  DrvLeonOSStartup(DDR_RESERVER_FOR_LAUNCHER_START );
}
```

This application should consider the following limitations:

- never overwrite the RESERVED DDR section (partially protected by linker because the linker makes a difference between DDR direct and DDR caches, which means, for example, 0x80000000 and 0xC0000000 will not be detected as overlapped sections)
- include USB stack and be able to go to a specific handler when receiving FU_CMD (a predefined FirmwareUpadate USB command).

Using the above allows us to rewrite the previously written firmware while we run App1, the AP wants to send a new `.elf` image and it will send an FU_CMD usb command. The App1 will jump to the

FU_CMDHandler() handler, set flag to 0 and start the UsbLoaderApp from DDR, which will take care of everything from now on. UsbLoaderApp will wait for the new `.elf`, write it to EEPROM, and execute it.



4 Final notes

The AP on the client side (Windows PC in this example case) should use the following pseudocode:

```
void M2FirmwareUpdate()
{
  sendUSBCommand(FU_CMD);
  wait(giveM2SomeTimeToSwitch);
  sendFirmware(Appx.elf);
}
```

To check FU_CMD format consult the documentation from the third party SW dev (MCCI).