VECTOR

Custom Flash Drivers

Version 1.0
2016-02-16
Application Note AN-ISC-8-1188

Author Alexander Starke

Restrictions Customer confidential – Vector decides

Abstract This application note describes the steps to be taken when embedding a custom flash driver into a Vector Bootloader

Table of Contents

1 Overview

The Vector Bootloader uses a standardized flash API, which has been developed by the HIS consortium. This standardized flash API offers the possibility to easily extend a Vector Bootloader by custom flash drivers.

Custom flash drivers might be used for various reasons:

- Adding support for updating additional external flash memory devices
- Adding support for updating additional external EEPROM devices
- Adding support for multi-processor flashing

Note
Vector offers implementation services for any of the above options. Please state your use-case and we will help you judge whether to implement a solution on your own or use our services.

This document is intended to provide OEM independent information. If the following explanations do not fit with your use-case please do not hesitate to ask for our support.

2 HIS Flash Driver Interface

A thorough description of the HIS flash driver interface can be found online. Please check section 0.

Typically your delivery is going to contain one or more HIS-compliant drivers which can be treated as a blueprint for an own implementation. Please check the following folders:

```
.\BSW\Fbl\fbl_flio.*
.\BSW\Flash\flashdrv.*
.\BSW\Eep
```

The flash API consists of the following functions:

```
IO_ErrorType FlashDriver_InitSync( void * );

IO_ErrorType FlashDriver_DeinitSync( void * );

IO_ErrorType FlashDriver_RReadSync( IO_MemPtrType, IO_SizeType, IO_PositionType );

IO_ErrorType FlashDriver_RWriteSync( IO_MemPtrType, IO_SizeType, IO_PositionType );

IO_ErrorType FlashDriver_REraseSync( IO_SizeType, IO_PositionType );

IO_ErrorType FlashDriver_VerifySync( void * );

IO_U32 FlashDriver_GetVersionOfDriver( void );
```

If you implement your own driver you will have to replace “Flash” with you own unique memory device descriptor in most cases to avoid name collisions.

The functions `InitSync()` and `DeinitSync()` implement the memory device initialization, which might include the initialization of additional peripherals, e.g. SPI or I²C.

The function `RReadSync()` is responsible to read the requested amount of data from the provided address into the given buffer. The Bootloader assumes that read accesses do not require any address alignment. If that is the case such behavior must be abstracted by this function.

Please note, that the function `RReadSync()` deserves special attention. Most, if not all, Bootloader flash drivers will implement the `RReadSync()` function in the flash driver wrapper (fbl_flio.*). This is to allow presence pattern or version checks during start-up, when a downloaded flash driver is not (yet) available. Nevertheless it is absolutely valid to solely implement the read logic in the downloadable part, if the previously mentioned use-case does not apply.

The `RWriteSync()` is responsible to actually write the provided data. When programming downloaded data the Bootloader memory stack will take care, that write address and length will always be correctly aligned (see also segment size description below). In other use-cases, e.g. when the driver is called by custom code, this may not be guaranteed. Hence the driver should ensure only correctly aligned address and length parameters are accepted. Additionally this function must verify that the provided buffer contents have actually been written, e.g. by implementing a read-back of the data.

The `REraseSync()` function shall erase the provided memory range. The Bootloader will provide erase regions with the segmentation configured in the flash block table in the configuration tool. If multiple physical flash sectors are grouped together in the flash block table this function is responsible to abstract the underlying hardware and erase all of them which are in range. Additionally this function shall verify that the requested memory range has actually been erased, e.g. by implementing a read-back of the data.

The function `VerifySync()`is a Vector extension to the API. `VerifySync()` is used if a 3rd-party flash API requires that a certain verify function is called once programming has finished.

The function `GetVersionOfDriver()` is required by some OEMs. For a statically linked driver this version is fixed, if the driver is downloadable this function shall provide the version information of the downloaded driver if it is available.

An important property of each memory device is the so called segment size. The segment size is the minimum number of bytes that has to be written at once. It has to be defined by each memory driver with a value that depends on the hardware characteristics.

```
#define FLASH_SEGMENT_SIZE     0x08u
```

Another important property is the erase (deleted) value of a memory cell. Most devices implement 0xFF but some hardware platforms deviate from the convention.

```
#define FBL_FLASH_DELETED      0xFFu
```

The string FLASH in the previous defines needs to be replaced by the name of your custom flash driver.

Depending on your environment (hardware platform, compiler) a flash driver might be either relocatable or non-relocatable.

Relocatable flash drivers can be downloaded into a RAM buffer which is linked to an arbitrary RAM address. The Bootloader enforces that such kind of flash drivers are always downloaded to address 0x00000000 (address based download). See chapter 4 for additional information regarding the RAM buffer handling.

For non-relocatable flash drivers the RAM buffer needs to be linked to the same address the flash driver is linked to. The Bootloader cross-checks the flash driver download address with the RAM buffer address (address based download).

Typically you do not need to set this attribute for your custom driver as it is inherited by the flash driver in your delivery.

```
#define FLASH_DRIVER_(NOT_)RELOCATABLE
```

3 Polling Function

3.1 General Information

Depending on the OEM and project requirements the Bootloader needs to execute certain tasks during flash programming, e.g.:

- Handle the watchdog

- Receive bus messages, e.g. the next transfer data (see also 3.5)
- Trigger RCR-RP messages (request correctly received, response pending)
- Answer diagnostic status requests

The Bootloader internally uses the function `FblLookForWatchdog()` to implement the above listed functionalities. In order to avoid any interferences of your flash driver with the Bootloaders core functionality please check that this function:

- Is unconditionally called by your driver while it is waiting for a hardware operation to complete (write, erase, SPI transfer …)
- Is called by your driver at a “reasonable rate” in each data processing loop

Caution
Implementing a custom flash driver without verifying that the call rate matches the projects requirements introduces significant risks. The Bootloader may work as expected on your desk, but might fail later on in the field. You should also take performance degradation into account, e.g. caused by flash memory aging.

You have to verify the call rate of `FblLookForWatchdog()` in your custom flash driver. Vector typically adds a GPIO pin toggle in `ApplFblWdTrigger()` which is expected to be called with a fixed interval (configurable in the configuration tool with a resolution of 1 ms). We observe the pin toggle using appropriate measurement hardware (e.g. a scope with statistical functions) to verify the implementation. Deviations from the expected toggle rate are documented in the test report of your delivery.

3.2 Internal Bootloader Time Base

The Bootloader has an internal 1 ms time base. Make sure to call `FblLookForWatchdog()` at least each 1 ms, otherwise the Bootloader is going to lose timer events. Be aware, that specific bus systems, e.g. FlexRay put further constraints on the required call rate.

Exception: Bootloaders which use a 3rd-party flash API (library) that does not allow fulfilling this rule. Those Bootloaders implement timer interrupts to ensure proper functionality.

3.3 Watchdog Handling

The watchdog trigger interval depends on your projects requirements. Check that the call frequency of `FblLookForWatchdog()` allows the Bootloader to fulfill those requirements. Especially window watchdogs might introduce constraints regarding watchdog trigger deviations.

3.4 (Adaptive) RCR-RP Handling

The RCR-RP handling in our Bootloaders has two implementation variants:

1st: Before starting a potentially long-lasting operation the Bootloader issues a RCR-RP and further RCR-RP messages are generated using the OEM-specific $P2_{Server}$ time.

2nd: The Bootloader issues RCR-RP messages only if required, which is called adaptive RCR-RP. This requires that all long-lasting operations are broken down into small time chunks. In between those chunks the Bootloader is able to check whether a RCR-RP message has to be generated. See also chapter 3.5.

Both variants only affect the write procedure, which is the processing of a UDS TransferData message. During erase we always use the 1st variant.

The 1st variant does not impose any additional constraints on the `FblLookForWatchdog()` handling (aside from those listed in 3.2).

The 2nd variant has some caveats. As already outlined the data to be written has to be broken into small chunks. The processing time of a single chunk shall not be larger than $P2_{Server}/2$. Please check the data sheet of your memory device for the worst-case write timing. The chunk size can be configured in the configuration tool and is called “write segmentation”. This size is currently used for all flash drivers embedded in the Bootloader. Its size must be aligned with a flash segment sizes.

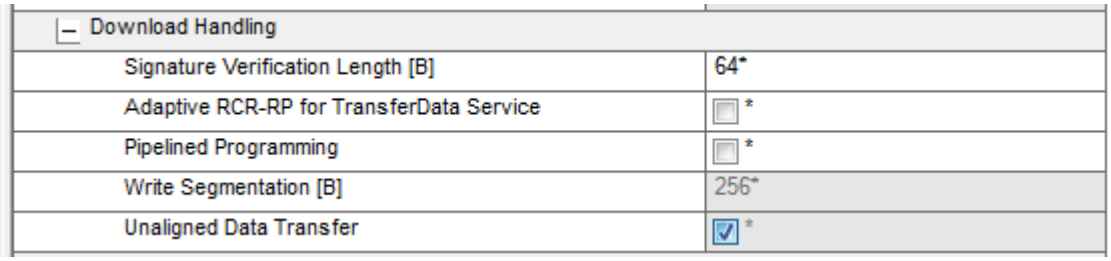

Figure 3-1 GENy

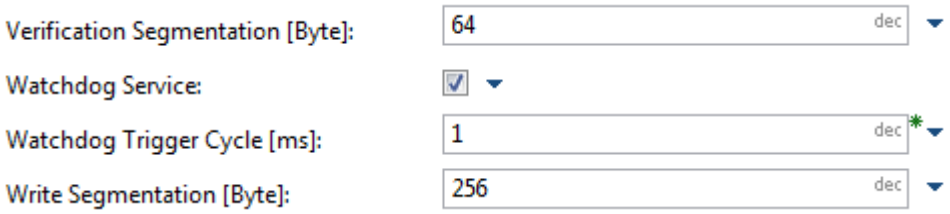

Figure 3-2 DaVinci Configurator

3.5 Pipelined Programming (Early Acknowledge)

The feature pipelined programming is available for some OEMs and platforms. When using pipelined programming the Bootloader optionally starts to receive the next UDS TransferData while it is processing (i.e. writing to memory) the previous TransferData, if there is enough spare time during data transfer. This is typically the case for slow bus interfaces or platforms with high flash memory performance. It improves the total update time of an ECU as the available bus bandwidth can be used more efficiently.

The implemented procedure is similar to the (adaptive) RCR-RP handling, as outlined in the previous section (3.4, 2nd variant).

3.6 Pipelined Verification

The feature pipelined verification is available for some OEMs and platforms. When using pipelined verification the Bootloader optionally starts to compute the checksum and/or hash over already written data, if there is enough spare time during data transfer. This is typically the case for slow bus interfaces or platforms with high computation performance. It improves the total update time of an ECU as the final verification step (e.g. RoutineControl CheckMemory) can be executed very fast as most work has already been done.

The basic procedure is similar to that used by (adaptive) RCR-RP handling and pipelined programming. The Bootloader takes small chunks of data and starts to compute a checksum and/or hash. In between the chunks the Bootloader checks if the next UDS TransferData message has been received. The chunk size is configurable in the generation tool, using the parameter “verification segmentation”. This size is currently used for all flash drivers embedded in the Bootloader. In most cases the read performance of a memory device does not deviate over its live time. The processing time of a single chunk shall not be larger than $P2_{Server}/2$.

4 Downloadable Flash Driver

Some OEMs specify a downloadable flash driver (also called secondary bootloader, SBL). In a few cases it might be useful to have a downloadable custom flash driver.

Advantages:

- Additional security, as the flash driver code is not located in the ECU during normal operation and might thus not unintentionally be executed.
- Flash driver can be updated in the field. If the flash driver is statically linked with the bootloader a bootloader update is the only way to realize an update. This is more complex and not specified by most OEMs.

Disadvantages:

- Increased complexity inside the Bootloader
- Increased RAM consumption, all downloaded drivers are stored in RAM
- Additional post-build steps might be required

Having a downloadable flash driver requires to split the logic into a flash driver wrapper (we call it fbl_flio) and the flash driver itself (flashdrv).

The tester will then download the flash driver contents during the update procedure. The flash driver wrapper is responsible to check if an appropriate flash driver is located in the temporary RAM buffer. Thereafter it can use the downloaded flash driver to modify the memory contents as requested by the tester.

There are a few possibilities how to use an additional custom flash driver together with the existing Vector flash driver. The most straight-forward solution is to reserve some margin at the end of the 1st flash driver and thereafter append the 2nd flash driver, typically in a post-build step. This adaptation is transparent for the delivered bootloader. Your custom flash driver wrapper must be aware of the relative offset, of course.

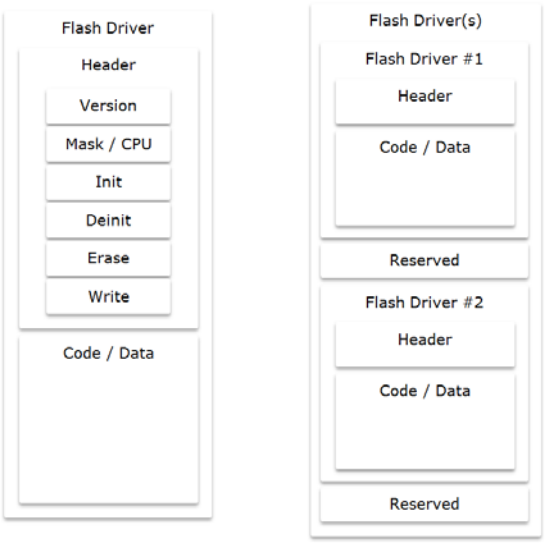

Figure 4-1 Flash Driver Layout

Note
This approach may be the only option, depending on the OEM. For some OEMs the Bootloader will only allow the further download, when all configured drivers (downloaded and statically linked) were initialized successfully.

Another option is to download the respective flash driver before downloading the actual data, which means having multiple flash driver containers. This procedure is only valid for a few OEMs.

5 Additional Resources

HIS Flash Programming (document overview)

HIS Flash Driver Specification v130

Flash Driver API v213

Vector Homepage - Flash Bootloader

Flash Bootloader Product Information

6 Contacts

For a full list with all Vector locations and addresses worldwide, please visit http://vector.com/contact/.