AN4808
Application note

Writing to non-volatile memory without disrupting code execution on microcontrollers of the STM32L0 and STM32L1 Series

Introduction

Microcontrollers often receive data that need to be stored or that require an immediate feedback, without having to stop the CPU activity, even for a few milliseconds.

However on some devices, the same memory controller is shared between code and data non-volatile memory (NVM). This may cause the program execution to stall while the are being written in the NVM.

Techniques to avoid this situation are described in this application note.

The X-CUBE-NVMRWW firmware package contains the project used as implementation example.

The following documents (all available from *www.st.com*) are to be considered as reference:

- Reference manual RM0038: “STM32L100xx, STM32L151xx, STM32L152xx and STM32L162xx advanced ARM®-based 32-bit MCUs”
- Reference manual RM0367: “Ultra-low-power STM32L0x3 advanced ARM®-based 32-bit MCUs”
- Reference manual RM0376: “Ultra-low-power STM32L0x2 advanced ARM®-based 32-bit MCUs”
- Reference manual RM0377: “Ultra-low-power STM32L0x1 advanced ARM®-based 32-bit MCUs”

Contents

1 Definitions 5

2 Memory interface summary 6

2.1 Timing 6

2.2 Bus stall 7

3 Implementation 8

3.1 Thread mode code 8

3.2 Interrupt handlers 8

3.3 Vector table 8

4 Alternative solutions 10

4.1 Using a dual bank device 10

4.2 Using an external EEPROM 11

5 Example projects 12

5.1 HW setup 12

5.2 Configuration options 12

5.3 Example operation 12

5.3.1 Download a file to EEPROM 13

5.3.2 Upload a file from EEPROM 13

5.3.3 Erase the contents of data EEPROM 13

5.3.4 Set/disable fixed EEPROM program timing 13

6 Conclusion 14

7 Revision history 15

List of tables

Table 1. List of acronyms 5
Table 2. Differences between internal and external EEPROM 11
Table 3. Revision history 15

List of figures

Figure 1. Memory interface simplified overview 6
Figure 2. Dual bank data access 10

1 Definitions

Table 1. List of acronyms

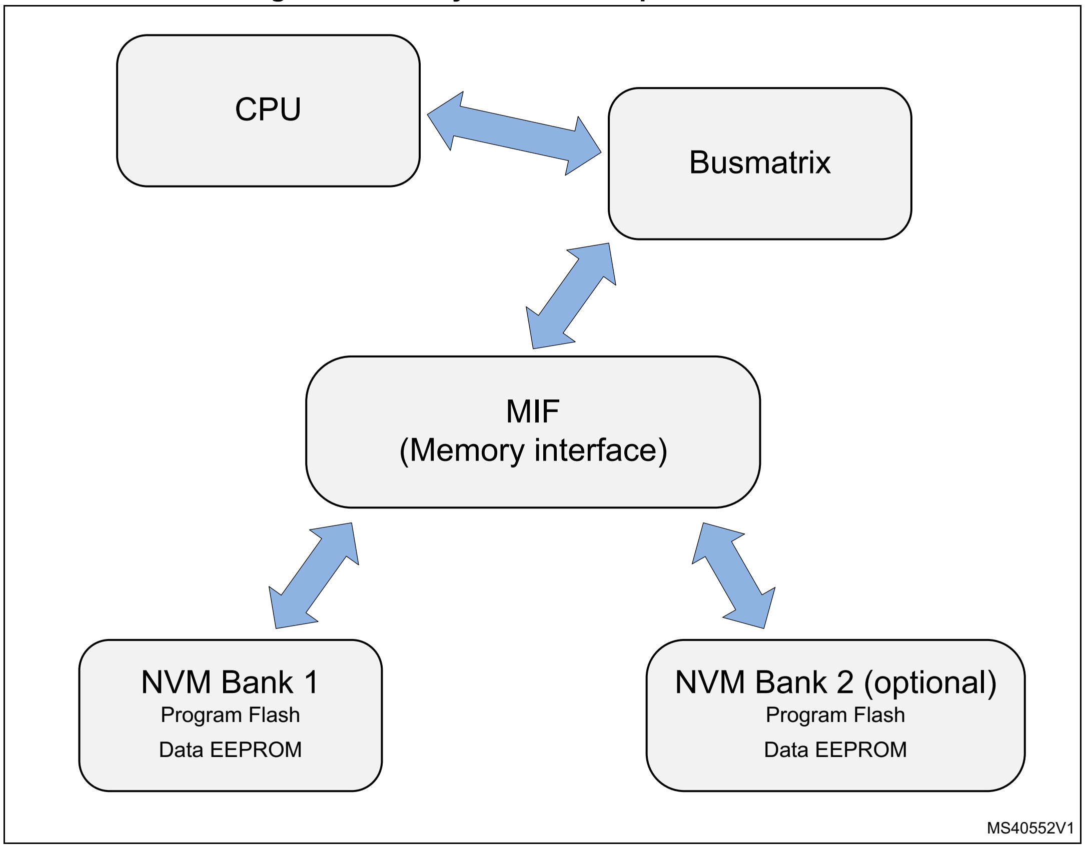
Term	Description
CPU	Central Processing Unit (part of the microcontroller)
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read Only Memory
I^2C	Inter-Integrated Circuit (industry standard serial bus)
IDE	Integrated Development Environment
MCU	Microcontroller
NVIC	Nested Vector Interrupt Controller
NVM	Non-Volatile Memory (EEPROM or Flash)
RM	Reference Manual
TC	Transmission Complete
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous and Asynchronous Receiver Transmitter

2 Memory interface summary

The NVM in the STM32L0 and STM32L1 Series is split in several regions, the most interesting for our case being the one featuring the Flash program memory and data EEPROM. Even though (at first glance) they may appear to be separate and independent, the program and data memory actually share the same memory controller (see *Figure 1*).

The memory interface is capable of reading both banks in parallel, or reading one bank while writing the other, however this parallelism is not possible within a single bank.

Figure 1. Memory interface simplified overview



2.1 Timing

EEPROM programming timing is based on a write/erase cycle duration. It is important to highlight the fact that some changes in memory only require one programming cycle, while others need two or more, this is described in detail in the cited reference manuals.

To keep the programming duration (t_{prog}) as short as possible, it is necessary to follow some simple rules:

- Only flip bits from reset to set, otherwise an erase cycle is automatically added before the write cycle
- Write only to aligned addresses (unaligned access needs to access two locations)
- On microcontrollers of the STM32L1 Series, do not activate the FTDW bit in FLASH_PECR register, on STM32L0 products the bit to stay clear is FIX in FLASH_PECR register.

If stalling the execution for one t_{prog} is acceptable, but more delay can cause a problem, it is possible to erase the target NV memory before writing.

Exact value of the EEPROM data write timing is available in the product datasheet, in this document we assume t_{prog} to be shorter than 3.33 milliseconds.

2.2 Bus stall

When the code is executing from the Flash program memory, the instruction must be read from it, this is not possible during a write operation on any memory block of the same bank. In such case the program execution stalls during the EEPROM data write.

Stalling happens at the Busmatrix interconnection and blocks the data recipient until the data is available.

The stall does not disrupt the sequence of operation. What happens is that the CPU core keeps waiting for the instruction fetch. Also, the stall only affects the CPU core. Processes independent from the core, such as DMA or peripherals may continue to work as long as they avoid accessing the NVM bank.

3 Implementation

When designing an application that utilizes the EEPROM data memory on single bank device, it is necessary to clearly identify the timing constraints and requirements.

There are basically two options:

1. Have the code necessary for any action that may occur during the write operation in the volatile memory (RAM)
2. Postpone the writing to a quieter moment, when no immediate action is necessary.

In this application note we assume that the second option is not applicable, hence the first one must be implemented.

It is necessary to initiate the data EEPROM writing from the RAM as well. When the execution is stalled due to fetch attempt from the busy NVM, neither events or interrupts are processed until the BSY flag is released. Having the interrupt code in RAM is not enough to wake the MCU.

3.1 Thread mode code

To prevent stalling the execution, thread mode code must be executed from the system RAM. The piece of code in RAM may be a function that initiates the writing and then loops while the BSY bit of the memory interface is set and takes care of the actions that must not wait until the write is completed.

3.2 Interrupt handlers

The interrupt handler code must be placed in RAM, at least in case of interrupt requests that must not wait until the EEPROM operation ceases.

The way to tell the linker to put the code in RAM differs according to each development environment, the user should refer to the IDE documentation.

Several practices must be avoided in the code in RAM:

- Do not call functions placed in program Flash memory from the system RAM code. Take extra care to avoid problems with the library functions and functions hidden in macros.
- Abstain from using constants that are placed in program Flash memory.
- Obviously, the data EEPROM cannot be read while being written.

When testing the code in RAM it is possible to gate the clock to the NVM and disable it (this is the simplest way to ensure that the critical code is indeed self-sustained in RAM).

3.3 Vector table

To maintain capability to process interrupts during the writing period, the vector table must be relocated to system RAM. By default the linker places the vector table at the beginning of the program memory. Its size (up to 192 bytes for the STM32L0 Series, up to 292 bytes for STM32L1 devices, for the exact size on any given product check the reference manual) must be allocated in the system RAM and table copied there from its original location.

The new interrupt vector table address must then be written in the System Control Block (SCB) register VTOR (Vector Table Offset Register). System control block is part of Cortex®- M0+ and Cortex®-M3 cores used, respectively, in the L0 and the L1 Series.

The vector table placement is limited to addresses aligned to number of vectors, extended to powers of two (each vector is 4 bytes). For L1, where there are 73 interrupt vectors, nearest power of 2 is 128, thus the size to align to is 128 x 4 bytes = 512 bytes. When attempting to write unaligned offset, the unaligned part of the address will be ignored, leading to (later) failure in execution.

It is also advised to check the vectors in the table and verify if those needed during data EEPROM write access period are pointing to RAM.

4 Alternative solutions

4.1 Using a dual bank device

Products with nonvolatile memory split in two semi-independent banks have a great advantage in this case, as one bank can be written while the code is executed from the other bank.

In case of the STM32L1 Series the code executed from the NV program memory typically runs 30% faster compared with it running from RAM (refer to AN 4777 “Implications of memory interface configurations on STM32L1 and STM32L0 Series microcontrollers”). This value, of course, depends significantly upon the memory configuration and the nature of the executed code. The SRAM execution performance with physical remap selected in SYSCFG memory remap register is better, while NVM interface speed depends on latency and prefetch.

In case of STM32L0 Series the differences tend to be smaller.

Figure 2. Dual bank data access

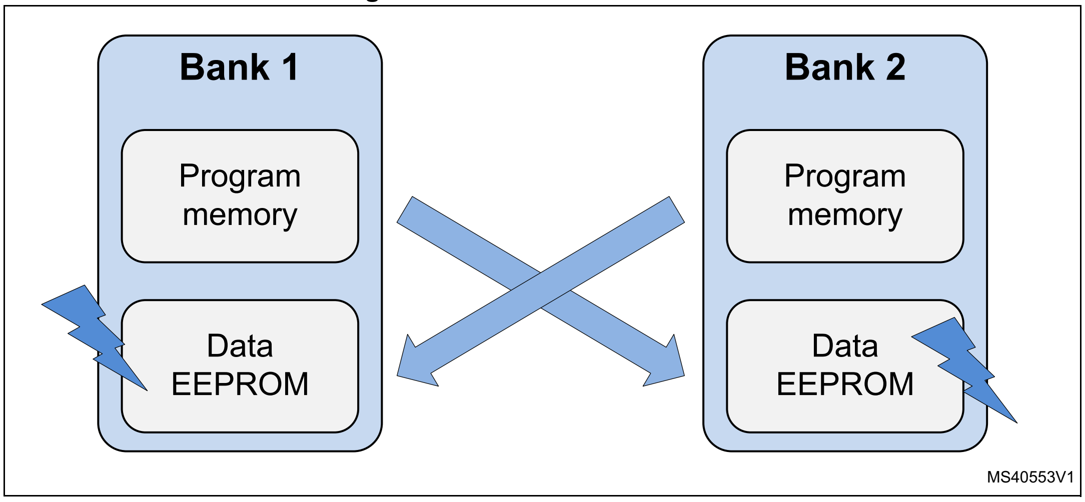


Figure 2 shows the ideal situation in which the code in bank 1 uses data EEPROM from bank 2 and code from bank 2 uses data EEPROM from bank 1 (a similar scenario is registered when program memory to be written is the one of the other bank).

Code and data placement must be designed with this optimal solution in mind, especially if code or data size exceeds the physical bank capacity.

Only Cat.5 devices from STM32L0 Series and Cat.4, Cat.5 and Cat.6 devices from the STM32L1 Series are equipped with two semi-independent banks of memory.

This solution has only one disadvantage, device selection is reduced to those having higher cost and/or larger packages.

4.2 Using an external EEPROM

External EEPROM connected by typically an I^2C interface may also serve the purpose.

However this solution has several disadvantages – more PCB space, extra cost, extra power consumption, communication overhead. Moreover, the write delay is not completely avoided, it's only delegated.

Table 2. Differences between internal and external EEPROM

Feature	External EEPROM (e.g. M24C64 I^2C serial EEPROM)	On-chip EEPROM
Write time (word)	20 ms	< 8 ms
Write method	CPU-independent, completely externalized once the data is sent.	May stall the program execution unless precautions are taken.
Read access (word)	92 µs	Immediate (one clock cycle latency may apply at high frequencies)

A look at *Table 2* may give the impression that the use of an external EEPROM is bringing more problems compared with those it solves, the positive aspect is that the I2C bus can be shared by several masters, and it also may store larger amounts of data.

5 Example projects

An example code is supplied with this application note, demonstrating the effects of different settings mentioned in this document and principles of execution from RAM.

This example configures a two word size circle buffer fed from the USART through a DMA channel. Such arrangement, along with portion of code available in RAM, ensures the stall-free execution. One word in the buffer can be received even during the time when the other word is being written in the EEPROM.

While it is possible to access the EEPROM by smaller data units (up to single byte), the t_{prog} remains constant, effectively reducing maximum possible communication speed (this is the reason why the example uses word access).

Use of DMA is particularly recommended in such cases, because even if (despite all the efforts) the program is stalled, DMA between RAM and the peripheral will continue transferring data. It effectively reduces risk of lost bytes and results in the lowest possible workload for the CPU.

The biggest difference between the two EVAL boards supported by the example software is in handling the situation where the data come at rates faster than those associated with the writing in the NVM. While the USART on STM32L0 microcontrollers is tolerant to overrun errors (they are ignored); on STM32L1 MCUs any such error will end the reception attempt.

5.1 HW setup

Connect USART2 port to a PC COM port and power the EVAL board. Use a terminal application that supports YMODEM protocol (Tera Term is offering a solid, cost free alternative to Windows® HyperTerminal).

5.2 Configuration options

In *main.c* two communication speeds are defined, namely COM_SPEED_SAFE and COM_SPEED_FAST.

The safe, slower speed of 4800 Bd should work regardless of the EEPROM state and timing settings. In this pace the time to transmit one word is approximately 6.66 ms, which is longer duration than two typical EEPROM write cycles. If single write cycle can be guaranteed it is possible to select COM_SPEED_FAST, which correspond to 9600 Bd. For the fast paced programming to succeed, the EEPROM must be cleared prior to file download and the fixed program timing must be disabled.

5.3 Example operation

With user interface provided by the terminal SW the operation is easy.

The example firmware first displays a header and a menu. Pressing numbers on PC keyboard selects one of the available choices.

5.3.1 Download a file to EEPROM

First menu choice initiates download of a file into the data EEPROM. The file size limit is constrained by the memory size. Select the file to download, it will be written during the communication process.

5.3.2 Upload a file from EEPROM

The second menu item is needed to get the data back from the data EEPROM and to compare them with the original to assess the communication success. As the file name and size is not stored, all the data EEPROM content is sent, resulting in a simple approach with a complete feedback.

5.3.3 Erase the contents of data EEPROM

For successful download of file at the fast data speed it is necessary to limit the writing time to single cycle. That requires erased data memory.

Select this option to erase the data EEPROM prior to programming during 9600 Bd communication.

5.3.4 Set/disable fixed EEPROM program timing

This setting toggles the eighth bit in FLASH_PECR register that fixes the programming timing. In case of the STM32L0 Series it is called FIX, on STM32L1 Series the name is FTDW. In both cases, when the flag is set, each programming is automatically preceded by an erase, regardless of the previous memory contents. This leads to more deterministic execution, but at the cost of efficiency and also memory durability. The bit is reset by default.

6 Conclusion

With careful design of the firmware it is possible to avoid necessity of using a dual-bank device or even an external EEPROM component to store non-volatile application data, while still keeping the ability to react to events in real-time applications.

There is also significant advantage in comparison with the solution where data are kept in RAM buffer and written only when communication ends. Data written directly to EEPROM are kept safe in case of power loss.

7 Revision history

Table 3. Revision history

Date	Revision	Description of changes
24-Mar-2016	1	Initial release.