AN229843

Memory Mapped Access to SPI F-RAM

Author: Gernot Hoyler
Associated Part Family: Cypress SPI F-RAM
Associated Code Examples: None
Related Application Notes: None

This application note compares classic EEPROM/Flash style access with fully memory mapped access in the context of SPI F-RAM devices. It discusses the advantages and disadvantages of both methods. Linux on i.MX8 is used as a test and benchmarking platform to shed some light on the performance aspects. The results show that memory mapped access has clear performance benefits and simplifies the application code compared to the classic serial EEPROM/Flash access method.

Contents

1 Introduction

Nonvolatile Cypress SPI F-RAM memories can be used in a variety of ways. First, their instruction set is compatible with classic serial EEPROM and Flash memories. This feature allows developers to operate F-RAM devices like an EEPROM or Flash part using existing software drivers.

On the other hand, F-RAM devices possess RAM characteristics and advantages: they can be read and written instantly on a byte-by-byte basis without the need for erasing or polling like Flash devices. Advanced modern SPI controllers can generate the required command sequences on-the-fly in hardware and support memory mapped access via pointers. This makes serial F-RAM devices look like normal RAM to the applications.

The two usage models are presented and compared in detail in the following sections.

2 EEPROM/Flash Style Access

If serial F-RAM is used like an EEPROM or Flash device, then the typical control flow is:

1. Open a special device file
2. Set the file offset to a certain position
3. Issue a read or write call.

Steps 2 and 3 are repeated as often as needed.

Adding F-RAM support to existing EEPROM/Flash drivers is usually simple. In many cases, it is enough to add just a new device ID to the list of supported devices in the driver source code to make the devices work. SPI commands to read and write data are compatible between EEPROM/Flash and F-RAM, and erase commands are simply ignored by the F-RAM device. Most applications do not rely on the default value of freshly erased memory (e.g. 0xFF) so this behavior is fine. In special cases where they do, the erased memory region can be explicitly set to the expected default

value by the erase function. In addition, polling code used in EEPROM/Flash software drivers to detect the end of program operations does not affect F-RAM. To such software drivers, F-RAM devices appear to be immediately done with any program or erase operation and control returns after a single polling iteration. Alternatively, polling might be completely disabled for F-RAM in the drivers.

In Linux, as a concrete example, the access method requires the user to open a Memory Technology Device (MTD) or EEPROM special device file and issue two system calls for every read or write. First, a call of `lseek()` to position the file descriptor to the desired offset and second issue either a `read()` or `write()` system call to read or write the data. For large blocks of data, the associated system calls and their overhead is insignificant and can be neglected. Throughput is the crucial parameter in such cases. For small data sizes (for example, variables of 1-16 bytes), however, the system call overhead causes noticeable latencies.

What makes things more complicated for applications is the need to allocate and manage buffers that are passed to the read and write functions. Very often, data is copied back and forth several times in this access method, to and from the buffers in the application and then again from the buffers to the SPI controller FIFOs in the device driver and vice versa. These copy operations have a negative impact on throughput on fast systems.

3 Memory Mapped Access

User managed data buffers and manual movement of data is not needed for memory mapped access (also known as Memory Mapped I/O or MMIO). In this access method, applications can read and write to F-RAM simply by dereferencing pointers to data objects of the desired size.

Software assistance is needed only during initialization to probe the device and later to set up an appropriate address mapping for the application. Once this mapping has been established, all read and write accesses run completely in hardware. This leads to a better performance level compared to classic EEPROM/Flash style access. Primarily, latencies are shorter resulting in significantly better results for small data sizes.

Furthermore, memory mapped access simplifies the code of applications. Data does not have to be copied back and forth between buffers, and system calls are not needed to access the F-RAM memory after initialization.

Finally, advanced features such as code execution directly out of SPI F-RAM (XIP) are only possible with a memory mapped setup. Although read-only applications are also possible with SPI Flash in a memory mapped setup, mapped writes fail on these devices due to their polling and erase requirements.

A challenge could be that controller specific setup code must be added to the software drivers. Generic driver code is hardly possible.

4 A Case Study

To investigate the performance benefits of memory mapped access, a NXP i.MX8QXP SoC with a Cypress Excelon Ultra CY15B104QSN F-RAM is used to provide a modern benchmarking platform.

The OS in this case is Linux (kernel 4.14.98) that runs the Cypress SPI Memories Driver stack version v19.4. This software driver supports both classic MTD as well as memory mapped access. The CY15B104QSN is operated in QPI mode at a SPI clock frequency of 100 MHz SDR. Thus, the maximum theoretical throughput for both read and write operations is limited to 50 MiB/s[1].

The i.MX8QXP FlexSPI controller supports memory mapped accesses via a small configurable table. This Look Up Table (LUT) can hold up to 32 sequences to synthesize SPI bus transactions on-the-fly in hardware. Index registers in the controller can be set to inform the processor which sequence(s) to execute for memory mapped read and writes, for example, if a pointer is dereferenced. It might be a single sequence or a set of multiple sequences, for example, if a Write Enable command plus a Program command has to be issued for a write operation. For QPI reads and writes to F-RAM, the following LUT entries/sequences can be used:

```
/* 4-4-4 Quad read sequence with 3 address bytes */
writel(LUT0(CMD, PAD4, SPI_QUAD_READ_HP) | LUT1(ADDR, PAD4, ADDR24BIT),
       base + FLEXSPI_LUT(SEQID_AHB_READ_444_3, 0));
writel(LUT0(MODE8, PAD4, 0) | LUT1(DUMMY, PAD4, flex->quad_spi_dummy),
       base + FLEXSPI_LUT(SEQID_AHB_READ_444_3, 1));
```

[1] MiB is used to denote 1,000,000 Bytes (vs. MB which is usually 1,048,576 Bytes).

```
writel(LUT0(FSL_READ, PAD4, 0),
       base + FLEXSPI_LUT(SEQID_AHB_READ_444_3, 2));
writel(0, base + FLEXSPI_LUT(SEQID_AHB_READ_444_3, 3));

/* 4-4-4 Program with 3 address bytes (two sequences) */
writel(LUT0(CMD, PAD4, SPI_WRITE_ENABLE),
       base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 0));
writel(0, base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 1));
writel(0, base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 2));
writel(0, base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 3));
writel(LUT0(CMD, PAD4, SPI_PROGRAM) | LUT1(ADDR, PAD4, ADDR24BIT),
       base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 4));
writel(LUT0(FSL_WRITE, PAD4, 0),
       base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 5));
writel(0, base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 6));
writel(0, base + FLEXSPI_LUT(SEQID_FRAM_WRITE_444_3, 7));
```

Note that CY15B104QSN has a sticky WREN (Write Enable) bit in the status register. Once this bit has been set, the device does not need explicit Write Enable commands anymore preceding every memory write operation. Thus, only the second sequence of the listed sequence pair for the write path is used.

Another optimization technique used is prefetching that can be done automatically by the i.MX8QXP FlexSPI controller. This feature affects and speeds up the read path for all access methods. It always loads data blocks of full 2 kB from F-RAM into some hardware buffers. Read requests from the software are then served out of these buffers.

Table 1 summarizes the measured results and shows the performance benefits of direct memory mapped access. In particular, latencies are much shorter compared to the standard Flash style access method (by more than 20x). The extremely short latencies leverage the instant nonvolatility feature of F-RAM and help in situations where system power is lost abruptly. Memory mapped access becomes a complimentary requirement in those instances, shortening the time window where data is at risk.

Table 1. Benchmarking Results for CY15B104QSN on i.MX8QXP

	Read Throughput	Write Throughput	Read Latency	Write Latency
Flash Style MTD Access	41.7 MiB/s	15.3 MiB/s	4.6 us	14.6 us
Memory Mapped Access	48.5 MiB/s	27.5 MiB/s	0.2 us	0.2 us

In this benchmark, throughput results are measured by reading or writing the entire device. For the memory mapped case, `memcpy()` is called to copy all main array data from F-RAM to normal system DRAM or vice versa. See Appendix A for some ARMv8-A specific `memcpy()` optimizations. With hardware prefetching disabled, read throughputs are of the same order as write throughputs.

Latencies denote the delay after a write or read operation has been issued by the software application until the data is physically transferred on the SPI bus. In this benchmark, latencies are measured by issuing small 1 byte read and write operations.

5 CPU Caching

By default, CPU caching is disabled on most platforms for the entire I/O memory space. This enforces ordered and uncombined memory accesses and is a must, for example, to fill hardware FIFOs or to program or erase Flash devices.

For F-RAM memories, however, CPU caches might be enabled in combination with memory mapped access to push the performance envelope further. With CPU caching, the natural burst size on the SPI bus for reads and writes is one cache line (64 bytes on i.MX8QXP). This makes better use of the available SPI bus bandwidth compared to a series of smaller transfers. However, during a power drop data might get lost if it resides in a cache line that has not yet been written back to F-RAM. Whereas for normal RAM memories this behavior is perfectly acceptable, for F-RAM it is not.

Enabling a simple read caching scheme (that is, with a write though cache policy) is safe for F-RAM, as data is written immediately back to the F-RAM array in this configuration.

If the application has clear synchronization points (for example, saving full camera images), then even a write back policy might be enabled. Smaller write operations can be combined with this scheme to build highly efficient full 64-byte cache line writes. However, barrier and cache maintenance instructions must be added to the synchronization points

of the source code, in this case to flush the cache from time to time. Such instructions cause data that has accumulated in the CPU cache to be explicitly written back, and thus eliminate the risk of data loss.

6 Conclusion

Most of today's SPI controllers support memory mapped access to external devices. Therefore, with these controllers, memory mapped access has become a viable option to consider and customers can benefit from it, specifically in case of F-RAM.

Memory mapped access to F-RAM has clear performance benefits and simplifies the application code compared to the classic serial EEPROM/Flash access method. It is universal, flexible, and integrates F-RAM seamlessly into a modern system.

By carefully analyzing and optimizing the application code, a combination of memory mapped access with CPU caching can further improve both throughput and latency.

7 Related Documents

Technical Documents	
ARM Cortex-A Series Programmer's Guide for ARMv8-A, Version 1.0, 2015	Describes the 64-bit ARMv8 architecture, its instruction set and cache behavior
i.MX 8DualXPlus/8QuadXPlus Applications Processor Reference Manual, Rev. D, 11/2018	Describes the NXP i.MX8QXP processor with its FlexSPI memory controller including all registers

Appendix A. Optimized 16-byte memcpy() for ARMv8-A

The default `memcpy()` implementation for ARMv8-A in Linux uses load-pair and store-pair assembly instructions that move two 8-byte registers at once. Unfortunately, these instructions trigger two 8-byte SPI bursts on the bus instead of a single 16-byte burst. To improve the situation, `memcpy()` can be optimized to use a 16-byte FP/SIMD register plus the corresponding load/store instructions, as shown below. This change creates the desired 16-byte SPI bursts on the bus.

```
memcpy16:
        sub     sp, sp, #48
        str     x0, [sp, 24]
        str     x1, [sp, 16]
        str     x2, [sp, 8]
        str     wzr, [sp, 44]
        b       .L8
.L9:
        ldrsw   x0, [sp, 44]
        lsl     x0, x0, 4
        ldr     x1, [sp, 24]
        add     x2, x1, x0
        ldrsw   x0, [sp, 44]
        lsl     x0, x0, 4
        ldr     x1, [sp, 16]
        add     x0, x1, x0
        ldr     q0, [x0]
        str     q0, [x2]
        ldr     w0, [sp, 44]
        add     w0, w0, 1
        str     w0, [sp, 44]
.L8:
        ldrsw   x1, [sp, 44]
        ldr     x0, [sp, 8]
        lsr     x0, x0, 4
        cmp     x1, x0
        bcc     .L9
        nop
        add     sp, sp, 48
        ret
```

Document History

Document Title: AN229843 - Memory Mapped Access to SPI F-RAM

Document Number: 002-29843

Revision	ECN	Submission Date	Description of Change
**	6835272	06/08/2020	New Application Note.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Code Examples | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709