An Introduction to SPI-NOR Subsystem

By
Vignesh R
Texas Instruments India
vigneshr@ti.com
About me

• Software Engineer at Texas Instruments India
• Part of Linux team that works on supporting various TI SoCs in mainline kernel
• I work on supporting peripheral drivers on TI SoCs, mainly QSPI, UART, Touchscreen and USB
• This presentation is mainly based on my experience of getting QSPI controllers on TI platforms to work in mainline kernel
What’s in the presentation?

• SPI-NOR flash and types
• Communication with SPI-NOR flashes
• SPI-NOR framework
• SPI-NOR controllers and types
• Writing a controller driver
• Ongoing work and what’s missing
What is a SPI-NOR Flash?

- Array of storage cells that behave like NOR gate → NOR flash
- Two types of NOR flash:
  - Parallel NOR
  - Serial NOR
- Serial NOR flash that is interfaced to SoC via SPI bus and follows SPI protocol → SPI-NOR Flash
- Reduced pin counts compared to parallel NOR
## Why SPI-NOR flash?

<table>
<thead>
<tr>
<th>Property</th>
<th>NAND</th>
<th>eMMC</th>
<th>SPI-NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>Upto 128GB</td>
<td>Upto 128GB</td>
<td>Upto 512MB</td>
</tr>
<tr>
<td>Bus width</td>
<td>x8/x16</td>
<td>x4/x8</td>
<td>x1/x2/x4/x8</td>
</tr>
<tr>
<td>Read speed</td>
<td>Slow random access</td>
<td>Similar to NAND</td>
<td>Fast random access</td>
</tr>
<tr>
<td>Write</td>
<td>Fast writes</td>
<td>Fast writes</td>
<td>Slower</td>
</tr>
<tr>
<td>Setup Requirements</td>
<td>ECC and bad block management</td>
<td>Needs tuning (for higher speed)</td>
<td>No overhead</td>
</tr>
</tbody>
</table>
Typical SPI-NOR flash

SPI Controller

SCLK
MOSI
MISO
WP#
HOLD#
CS#

SPI-NOR Flash
Multi IO Flash

There are:
Dual IO, Quad IO and Octal IO flashes
SPI-NOR Flash Hardware

• Flash is composed of Sectors and Pages
• Smallest erasable block size is called Sector
  – May be 4/32/64/256 KB
• Sectors sub-divided into Pages
  – May be 256/512 bytes
  – Flash program is usually in page size chunks (though not necessary)
• Need to send Write Enable(WREN) command before a write or erase operation
• Most flashes support Read ID(RDID) command for flash detection
Communication Protocol

- **Command Phase** (1 byte)
- **Address Phase** (3/4 bytes)
- **Wait Phase** (n cycles)
- **Data Phase** (n bytes)
Types of Controllers

• Traditional SPI controllers
  – Provide direct access to SPI bus
  – Are not aware of the connected SPI slave device
  – Normally does not have deep FIFOs

• SPI-NOR Controllers
  – Aware of flash communication protocol (command, address and data phase)
  – Low latency access to flash, read pre-fetch and large HW buffer
  – May not provide direct SPI bus access

• Specialized SPI Controllers
  – Support both traditional SPI devices and Flashes
  – Typically, provides accelerated SPI-NOR access
SPI-NOR Framework

• Merged in v3.16
• Under Memory Technology Devices (MTD) Subsystem:
  – drivers/mtd/spi-nor/spi-nor.c
• Derived from pre-existing m25p80 flash driver code
• Why SPI-NOR framework?
  – Support controllers that only support flash slave devices
  – Support SPI-NOR/Specialized SPI controller hardware
    • Know flash specific data like opcodes, address width, mode of operation etc
  – Detect connected flash and choose suitable protocol for read/write/erase
Traditional SPI Controller

CPU/DMA

TX FIFO

CPU

Config Regs

RX FIFO

Shifter

SPI SCLK

CS

Data

SPI-NOR Flash

CPU/DMA

TX FIFO

CPU

Config Regs

RX FIFO

Shifter

SPI SCLK

CS

Data

SPI-NOR Flash

CPU/DMA

TX FIFO

CPU

Config Regs

RX FIFO

Shifter

SPI SCLK

CS

Data

SPI-NOR Flash
Accessing flash via SPI framework

- MTD layer abstracts all type of raw flash based devices like NAND, NOR and similar devices.
- Provides char(/dev/mtdX) and block(/dev/mtdblockX) device support
- Abstracts flash specific properties like sector, page and ECC handling
- Wear and bad block handling using UBI
- Handles partitioning of flash storage space
- /proc/mtd lists all devices
Accessing flash via SPI framework

- Handle SPI-NOR specific abstractions
  - Implement read, write and erase of flash
  - Detect and configure connected flash
  - Provide flash size, erase size and page size information to MTD layer

- Provides interface for dedicated SPI-NOR controllers drivers
  - Provide opcode, address width, dummy cycles information

- Support Multi IO flashes
Accessing flash via SPI framework

- Translation layer between SPI-NOR framework and SPI core
- Convert command, address and data phases into `spi_transfer` structs based on data that is supplied by SPI-NOR (via `spi_nor` struct)
- Generate `spi_message` objects for `spi_transfer` and submit to SPI core for read/write or other flash operations
Accessing flash via SPI framework

- SPI core validates, queues and sends SPI messages from upper layer to controller drivers
- SPI controller driver writes data to TX FIFO and reads data from RX FIFO
- Does not distinguish transfers as command or data or address
SPI-NOR controller-MMIO interface
Accessing flash via SPI-NOR framework

- SPI-NOR layer provides information about the connected flash
  - Passes `spi_nor` struct:
    - Size, page size, erase size, opcode, address width, dummy cycles and mode
- Controller configures IP registers
- Controller configures flash registers as requested by framework
- Controller drivers implements reads and writes
  - MMIO interface or from internal HW buffer
Specialized SPI controller-MMIO interface

- Memory mapped interface
  - Addr: 0x8000000
  - Addr: 0xFFFFFFFF
- SRAM
- Flash Command Generator
- IP Regs
- Config interface
- Direct access path
- TX FIFO
- RX FIFO
- SHIFTER
- SPI SCLK
- CS
- DATA
- QSPI-NOR FLASH
- Texas Instruments
Specialized SPI controllers with MMIO support

- SPI flash is configured using m25p80 and regular SPI interface
- Usually writes and erase operations are also done via SPI regular interface using `spi_message` struct
Specialized SPI controllers with MMIO support

- Flash read operation is done via MMIO interface.
- m25p80 driver calls spi_flash_read() API of SPI core
- Drivers of SPI controller with MMIO interface implement spi_flash_read()
- spi_flash_read_message struct provides info related to flash
Where to put your driver?

- Supports any type of SPI device and direct access to bus
  - Use SPI framework
- Supports only SPI-NOR flashes and optimized for low latency flash access
  - Use SPI-NOR framework
- Supports all SPI devices and has special interface for flash
  - Use SPI framework and also implement `spi_flash_read()` interface
Writing a SPI-NOR controller driver

• Following four callbacks need to be implemented:

  int (*read_reg)(struct spi_nor *nor, u8 opcode, u8 *buf, int len);
  int (*write_reg)(struct spi_nor *nor, u8 opcode, u8 *buf, int len);
  ssize_t (*read)(struct spi_nor *nor, loff_t from, size_t len,
                  u_char *read_buf);
  ssize_t (*write)(struct spi_nor *nor, loff_t to, size_t len,
                  const u_char *write_buf);

• Call spi_nor_scan() to ask SPI-NOR framework to discover connected flash

• Then call mtd_device_register()
Cadence QSPI DT fragment

qspi: qspi@2940000 {
    compatible = "cdns,qspi-nor";
    #address-cells = <1>;
    #size-cells = <0>;
    reg = <0x02940000 0x1000>,
         <0x24000000 0x4000000>
    interrupts = <GIC_SPI 198 IRQ_TYPE_EDGE_RISING>;
    flash0: flash@0 {
        compatible = "jedec,spi-nor";
        reg = <0>;
        spi-max-frequency = <96000000>;
    };
    flash1: flash@1 {
        ...
    };
};
## Performance Comparison

<table>
<thead>
<tr>
<th>Parameter</th>
<th>SPI transfers</th>
<th>SPI-NOR controller driver</th>
<th>SPI core’s flash read interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Speed</td>
<td>800 KB/s</td>
<td>4MB/s</td>
<td>4MB/s</td>
</tr>
<tr>
<td>CPU Load</td>
<td>~70%</td>
<td>~100%</td>
<td>~100%</td>
</tr>
<tr>
<td>Read with DMA</td>
<td>No HW support</td>
<td>No support in framework</td>
<td>20MB/s (15% CPU load)</td>
</tr>
<tr>
<td>Write Speed</td>
<td>400KB/s</td>
<td>400KB/s</td>
<td>400KB/s</td>
</tr>
</tbody>
</table>

Using TI QSPI controller on DRA7 SoCs under different framework with SPI bus rate of 64MHz
Ongoing work

• Choosing the right opcode based on controller and flash capabilities
  – Making sure communication with flash is stateless
  – Use opcodes that support 4 byte addressing

• Choosing 1-1-4 or 1-4-4 or 4-4-4 mode
  – Quad Enable (QE) bit behavior is different on different flashes
    • Spansion supports (1-1-4 and 4-4-4) but Micron supports only (4-4-4)

• Handling different sector sizes
  – A Flash may support 32K/64K/256K sector and optionally 4K sectors

• Serial Flash Discoverable Parameters(JESD216) and Basic Flash Parameter Table Support (merged in v4.14)

• Octal mode and DTR mode support
Adding DMA support

- Flash filesystems and mtdblock are not written with DMA in mind
  - Uses vmalloc’d buffers
  - Known to cause problems with VIVT caches
  - Buffers backed by LPAE memory are not accessible by DMA engines

- One solution is to use bounce buffers
  - Drivers like TI QSPI use bounce buffers

- SPI core has its own vmalloc buffer to sg_list mapping logic
  - Individual framework/drivers have own implementation

- Can DMA Mapping APIs be modified to map vmalloc’d for DMA for wider community benefit?
  - Provide bounce buffer, if mapping is not possible
References

- Various Flash datasheets: Micron, Spansion and Macronix
- JEDEC Standards: JESD216, JESD216A and JESD216B ([www.jedec.org](http://www.jedec.org))
- MTD mailing list ([linux-mtd@lists.infradead.org](mailto:linux-mtd@lists.infradead.org)) archive.
- [https://git.kernel.org/](https://git.kernel.org/)
Credits

• Texas Instruments Inc.
• The Linux Foundation
Q & A

Thank You!