Embedded Masters

# Adesto Serial Flash Demo Kit: Flash Driver Porting Guide

**Introduction:**
This document will provide an overview to porting the Adesto Flash Drivers to other platforms other than the Silicon Labs EFM32. This document will cover the primary files that need to be taken into consideration during the porting process. Each file is analyzed and individual functions outlined for each file.

**Key Source Files:**

***spiflash.c*** – This file contains all the Adesto Flash Drivers that are needed for porting. These are developed as Top Level firmware drivers that can be ported to other MCU/SOC platforms.

***spiflash.h*** – This file contains the Adesto Flash Driver function prototypes along with some defines and structure definitions.

***spi.c*** – This file contains specific SPI functions and initializations. Many of the functions in this file are specific to the EFM32. These EFM32 specific functions would be replaced by the MCU specific SPI functions that would be applicable to the same functions in this file. There are some functions that can remain in the porting process. These will be outlined later.

***spi.h*** – This file contains the function prototypes for the functions in spi.c along with various define statements.

***low_power.c*** – This file contains the functions to put the MCU/EFM32 into various low power states. The Low power states would need to be replaced with your specific MCU low power modes that would be comparable. These are further explained later in the document.

***low_power.h*** – Contains function prototypes for low power functions.

**Optional Software:**
Silicon Labs Simplicity Studio – Can be used to compile the demo software and program onto an EFM32 to run the Adesto Serial Flash Demo with the EMSENSR-WSP and the various Embedded Masters Adesto Memory Breakout boards.

**Supporting Documentation:**
Users can also refer to the Adesto Serial Flash Demo Kit documentation generated from the Doxygen comments in the source code. Users may also want to reference the EFM32 Leopard Gecko Datasheet and/or Reference Manual.

**spiflash.c/spiflash.h**

This file in general can be directly ported to your own specific project. All drivers required for specific Adesto commands/functionality are contained in these files.

**Command Definitions:**

If you want to add additional commands they can be entered into the #defines at the beginning of the file at the location shown below.

```
/* Adesto Command Hex Values */
#define CMD_READ_ARRAY                  0x0b
#define CMD_READ_ARRAY_SLOW             0x03  /* no dummy byte, but limited to
lower clock rate */
#define CMD_READ_ARRAY_DUAL             0x3b  /* not used, as we don't have dual-
SPI hardware */
#define CMD_PAGE_ERASE                  0x81
```

Append additional commands as desired…

**Adding additional Devices:**

Additional devices you may want to use can be added to the spiflash_info_table[] array which is an array of spiflash_info_t structures as defined in spiflash.h. New devices can be added just as the devices that are currently in the table have been defined. Items such as the Flash Size, Device ID, Program/Erase Sizes, etc have been defined. It should be noted that Fusion Devices that have the SO Interrupt capability should have the .has_so_irq set to true. An examples of one of these structures is shown below for the AT25XE021.

```
    [AT25XE021A] = {
          .name                     = "AT25XE021A",
        .id_size                  = 4,
        .id_bytes                 = { 0x1f, 0x43, 0x01, 0x00 },
        .device_size              = (2 << 20) / 8,
        .address_bytes            = 3,
        .program_page_size        = 256,
        .erase_info_count         = 5,
        .erase_info               = {{ 256,    CMD_PAGE_ERASE,          true },
                                     { 4096,   CMD_BLOCK_ERASE,         true },
                                    { 32768,  CMD_BLOCK_ERASE_LARGE,   true },
                                    { 65536,  CMD_BLOCK_ERASE_LARGER, true },
                                    { (2 << 20) / 8, CMD_CHIP_ERASE,  false }},
        .protection_sector_sizes = at25xe021a_protection_sector_sizes,
        .protection_sector_count = sizeof(at25xe021a_protection_sector_sizes) /
         sizeof(size_t),
        .read_status_cmd          = CMD_READ_STATUS,
        .status_busy_mask         = 0x01,
        .status_busy_level        = 0x01,
        .has_so_irq               = true,
        .so_done_level            = 0,
        .dataflash                = false,
    },
```

### SPI Command Functions:

These functions to write commands to the Serial Flash devices all make use of the spi_xfer function(spi.c) which is a pretty generic/portable spi transfer function outside of the fact that the user would need to update various items such as the SPI and low power mode specifics. This is further detailed in spi.c/spi.h section of this document.

**spiflash_single_byte_command** -> Used to send single byte command to the serial device.
**spiflash_multiple_byte_command** -> Used for multi-byte commands.
**spiflash_command_with_address** -> Used for commands that require an address to be sent.

### SPI Read Command:

spiflash_read -> This command is used to read data from the Serial device.

### SPI Erase/Write Commands:

Both the Erase and the Write commands follow the same structure in which there is essentially a state machine that moves the erase/write commands through the entire process. The Erase/Write commands each have completion states 1-5 with 1 being the first state and 4 or 5 being the final state depending on whether the device has the SO Interrupt enabled(if applicable, only on Fusion Devices). An pseudo example is shown below for how this occurs for this functionality

***spiflash_erase/write*** is called

- Manages completion states
- Determines if Active SO is enabled or not
- Checks address alignment
- Puts MCU into low power mode during erase
- Initializes state machine by specifying completion1(initial state)

***completion1***

- Configures Write Enable
- Increments state to completion2

***completion2***

- Issues erase/write command.
- Checks address alignments for erase/write addresses specified.
- Increments state to completion3

***completion3***

- Advances the address/decreases the length counter.
- Checks if Active SO is enabled(Fusion Serial Flash Devices)
  - YES – Send Active SO command and increment state to completion 4

- NO - Set state to completion5

***completion4***

- Calls ***spi_wait_so()*** function which enables the GPIO interrupt on the SO line and checks to see if the SO level has changed
  - Puts the MCU into EM2(Ram is alive, ~1.5uA)
- Increments state to completion1 to start the process again if needed.

***completion5***

- Check the Busy flag to determine if erase/write has completed.
- Sets state to completion1

### DataFlash RMW(Read-Modify-Write)

These commands are specific to the Dataflash(DB) devices. They perform similar to the Erase/Write commands with completion states 1-3.

There are other commands included in the spiflash.c file that hand various things such as reading the DeviceID, Reading the Status Byte/Bytes, Adesto Flash Power Down modes, etc.

### spi.c/spi.h

These files contain the lower level SPI Drivers. Most are specific to the EFM32. The functions that are highly portable are outlined below.

#### spi_xfer:

This is a fairly generic SPI transfer function that can be ported with the following modifications.

```
GPIO_PinOutClear(CS_PORT, CS_PIN);  // assert CS
```

Modify this line to handle your specific Chip Select Pin and use your specific GPIO Pin clear function.

```
// enable interrupts to start transfer
SPI_PORT->IEN |= (USART_IEN_TXBL | USART_IEN_RXDATAV);
```

This line enables the interrupts for the SPI module. Modify to your specific MCU SPI Interrupt Enable command.

```
EMU_EnterEM1();
```

This command puts the EFM32 into an IDLE mode where the core is shut-down but all peripherals and clocks can remain active. This is done to minimize application Power it could be bypassed but any MCU that has power modes should have a similar 'IDLE' type of mode that can be used here.

### SO_IRQ:

This is the SO/MISO Interrupt handler. This function is highly portable the GPIO control and interrupt commands shown below should be changed to your specific MCU pins/functions.

```
int p = GPIO_PinInGet(RX_PORT, RX_PIN);
```

Change to your specific MISO pins. This function gets the Pin level.

```
// disable IRQ
GPIO_IntConfig(RX_PORT, RX_PIN,
                  false,   // risingEdge
                  false,   // fallingEdge
                  false);  // enable
```

This function temporarily disables the interrupt on the MISO line.

```
GPIO_PinOutSet(CS_PORT, CS_PIN);  // deassert CS
```

This function sets the CS pin high. Change to your specific CS pin and use your MCU specific GPIO Pin Set command.

### spi_wait_so:

This is the function that is called from either the Erase/Write completion4 state if Active SO is enabled. It is portable with the following modifications for your specific MCU pins and GPIO control functions.

```
INT_Disable();
```

Disables interrupts temporarily. Modify to your specific MCU Interrupts Disable command.

```
p1 = GPIO_PinInGet(RX_PORT, RX_PIN);
```

This function checks the level of the SO/MISO pin. Modify to your specific SO/MISO pin and GPIO 'Get GPIO Level' function.

```
GPIO_IntConfig(RX_PORT, RX_PIN,
                          level,  // risingEdge
                          !level, // fallingEdge
                          true);  // enable
```

This function configures the SO/MISO Interrupt to occur on a rising edge. Modify to your own specific SO/MISO pins and specific MCU Edge-Level Interrupt enable commands.

```
INT_Enable();
```

This command re-enables any configured interrupts. Change to your MCU specific Global Interrupt enable command.

```
enter_low_power_state();
```

This command puts the EFM32 into a Low Power State. This function is outlined in lowpower.c/lowpower.h.

**lowpower.c/lowpower.h**

These files contain simple functions for putting the EFM32 into low power modes. The specific modes that are entered will be MCU specific and should be tailored to your specific MCU. The primary function is shown below and the description of the Low Power Modes so that you can choose the appropriate Low Power Modes for your specific MCU.

**enter_low_power_state():**

This function does a simple check to see if the SPI bus is active. Based on this it either puts the EFM32 into either EM1 or EM2 Energy Modes for the EFM32. These energy modes are further described below.

**EM1:** This is essentially an IDLE mode. The Core is shut-down but all other clocks/memory/interrupts are enabled.

**EM2:** This mode is similar to many DEEP SLEEP modes on other MCU's that keep the RAM Enabled, Low Power Clocks/Peripherals can be active, and all Interrupts are enabled. This is a ~1uA Energy Mode.

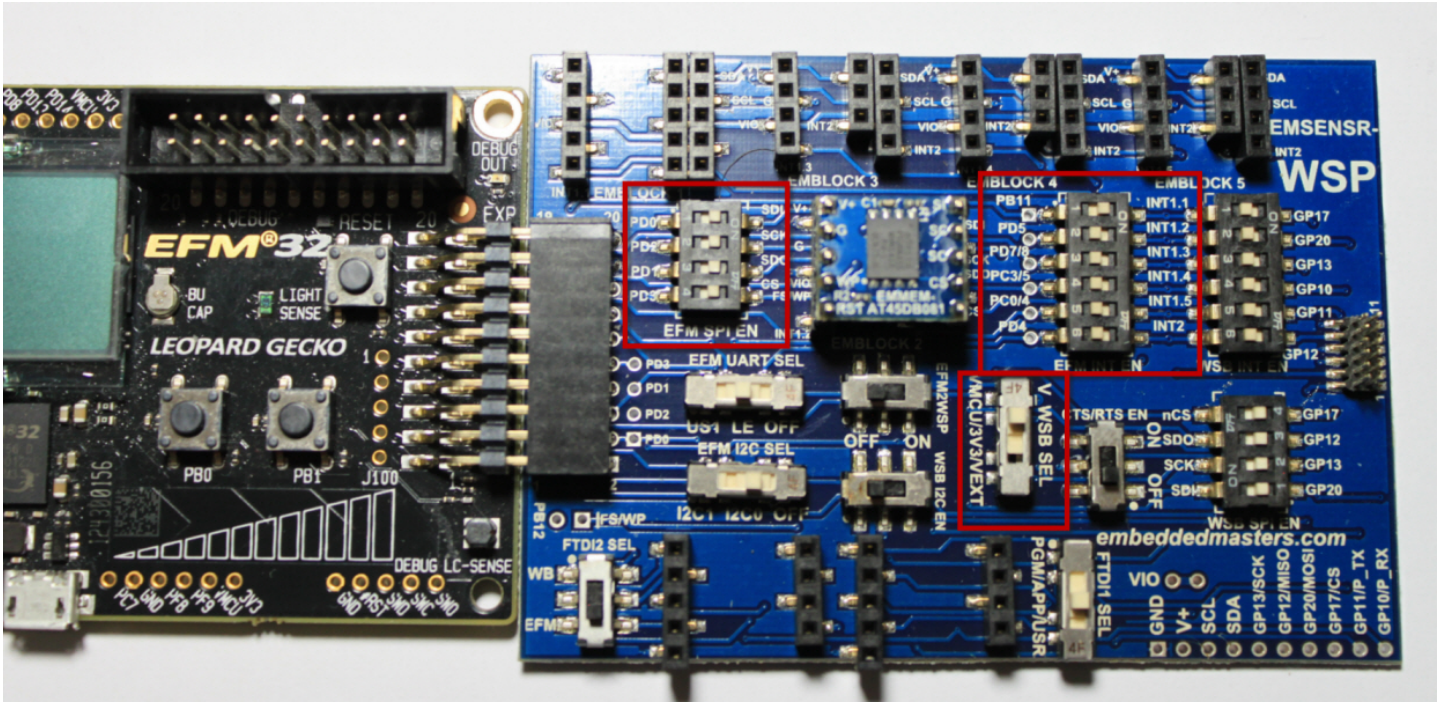
**Conclusion:**

This document should provide you the necessary understanding of how to port the Adesto Serial Flash drivers contained in the Adesto Serial Flash Demo to your own specific MCU platforms. If you have specific questions you can inquire with Embedded Masters LLC.

## Appendix A

### Demo Brief:

Brief outline/picture of the Adesto Serial Flash Demo with the EFM32 LG Starter Kit and the Embedded Masters EMSENSR-WSP plugin board along with the EMMEM-AT45DB081 plugged into the EMBLOCK2/SPI slot. This picture shows the correct pin settings for the demo.




Note: Picture Shown with EMMEM-AT45DB081

A. **EFM SPI EN** -> All selections set to ON. These connect the SPI lines on the EMSENSR-WSP to the EFM32 MCU Pins that are configured in software for SPI.
B. **V_WSP SEL** -> Ensure Slide switch is set to uppermost position/VMCU. The Energy Profiler monitors the VMCU voltage line provided by the STK3600.

For a full outline of the demo users can refer to the Adesto Serial Flash Demo: Quick Start Guide available on [www.embeddedmasters.com](http://www.embeddedmasters.com)