

# Technical Note

## Enabling Second Source Devices in the Micron® FDI Low-Level Driver

### Introduction

The Micron® Flash Data Integrator (FDI) low-level driver is the interface between the FDI software and the hardware Flash memory component. This document provides instructions for porting a second source device into the FDI low-level driver.

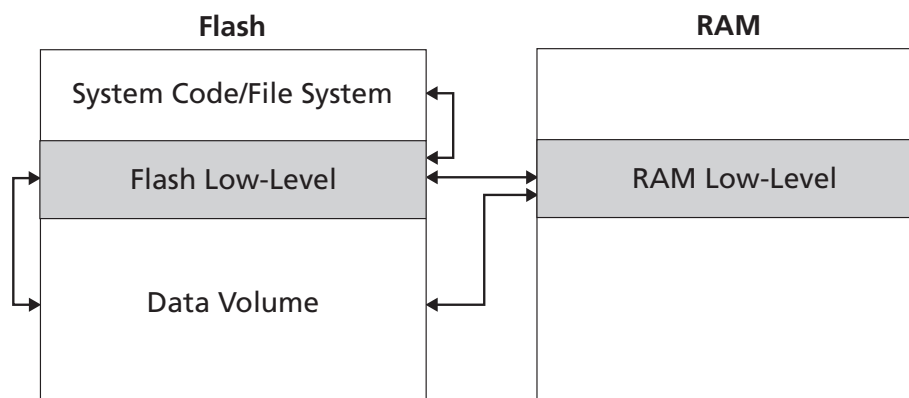
Micron's FDI low-level driver is the interface between the FDI software and the hardware Flash memory component. This document provides details for porting a second source device into the FDI low-level driver.

### Low-Level Architecture

FDI's low-level architecture (Figure 1) provides the following capabilities:

- Flash low-level
  - Aligns data and pads buffers before sending data to RAM low-level for data writes
  - Calls RAM low-level to erase blocks
  - Reads directly from the Flash memory device
- RAM low-level (single partitioned only)
  - Issues PROGRAM/ERASE command sequences to the Flash memory device
  - Checks for hardware errors
  - Provides interrupt polling

**Figure 1: Low-Level Architecture**

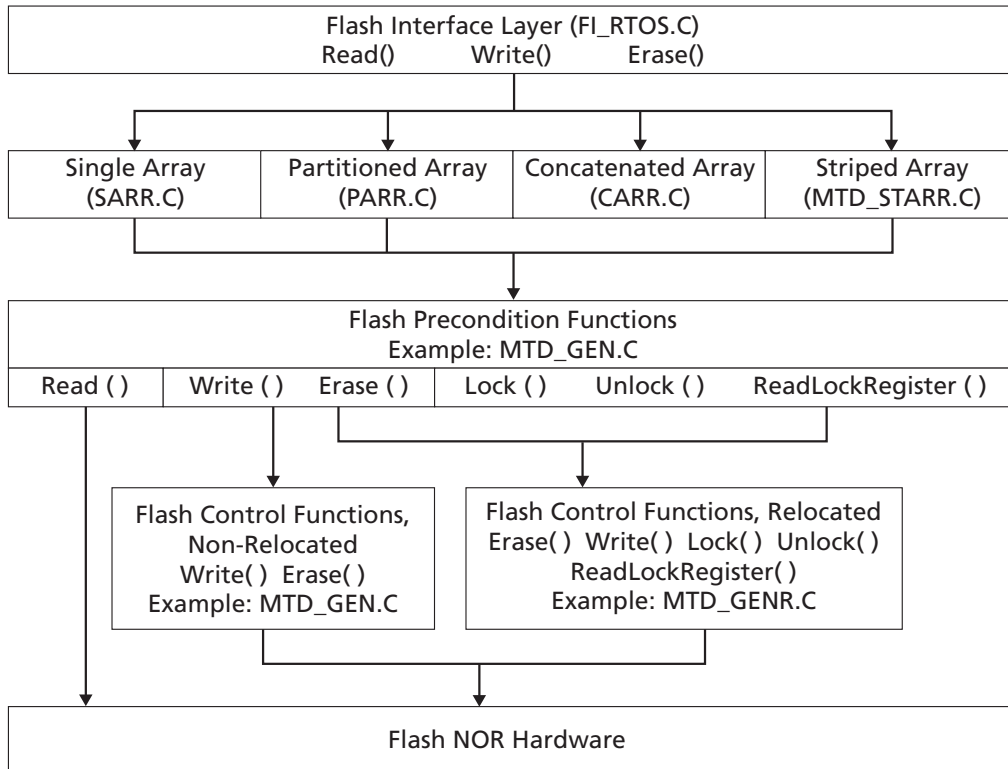


The low-level driver performs all READ, WRITE, and ERASE access. The low-level Flash memory driver functions (which might be relocated code based on your configuration) are located in separate files. These files might also require modification based on your specific system.

When in a single partitioned part with code executing in parallel any subroutines that the low-level driver calls internally cannot be called through a relative branch. Because some low-level functions are copied into RAM, all relative addressing is invalid. Also, operating system routines stored in Flash memory might not be called while interrupts and task scheduling are disabled.

Figure 2 shows the low-level architecture.

**Figure 2: Low-level Block Diagram**



## Setting up the Low-level Driver

The following instructions provide high-level steps for setting up the low-level driver. Cross-references are provided to detailed information where applicable.

1. Update the MTD initialization flow (see “Low-Level Initialization” on page 3).
2. Create the following runtime functions:
  - *SecondSourceFlash\_Read*: Reads information from the Flash memory device.
  - *SecondSourceFlash\_Write*: Writes information to the Flash memory device.
  - *SecondSourceFlash\_Erase*: Erases information from the Flash memory device.
  - *SecondSourceFlash\_Lock*: Locks a specified Flash memory device.
  - *SecondSourceFlash\_Unlock*: Unlocks a specified Flash memory device.

For more information, see “Precondition Functions” on page 12.

## Low-Level Initialization

This section describes the key phases of the low-level initialization. There are two key phases of the initialization:

- Gather the device information.
- Initialize the FlashDevice structure.

## Compile Option Method

Using a # define, disable the call to *MTD\_GetFlashDeviceInfo*. This function parses the Micron common Flash interface (CFI) on the device and populates a *FLASH\_DeviceInfo* structure. The data in this structure is used to set up the *FlashDevice* structure and perform simple error checking.

**Note:** Code in *italics* are suggested code changes for enabling a second source device in the existing code.

The following code is located in the *CreateDevices* function in *MTD\_INIT.c*.

```
if (current_element->OEMSubArrayPtr->Type == SUBARRAY_TYPE_NOR)
{
    #if (SecondSource_NOR == TRUE)
        device_info.PrimaryCommandSet = SS_CMDSET_XX
    #else
        status = MTD_GetFlashDeviceInfo(&device_info,
            (VOID_PTR)FlashQueryAddress,
            (VOID_PTR)FlashQueryAddress,
            buswidth);

        if (status != FLASH_ERR_NONE)
        {
            break;
        }
    }
```

```
num_dev_on_bus = (UINT8)
    (1<< (device_info.Configuration &
FLASH_ConfigGroupingMask));

/* check to make sure whole device has been given.
 * does device size not less the total length in the linked
list.
 */
current_flash_length =0;
for (test_token = current_element; test_token != NULL;
    test_token = test_token->Next)
{
    if( test_token->OEMSubArrayPtr == NULL )
    {
        continue;
    }
    current_flash_length += test_token->OEMSubArrayPtr->SubArray-
Length;
}
if (current_flash_length !=
    (UINT32) (1<< (device_info.DeviceSizeExponent +
    (num_dev_on_bus>>1))))
{
    status = FLASH_ERR_DEVICE_SIZE_MISMATCH;
    break;
}
#endif

/* create the device based on the primary command set */
switch (device_info.PrimaryCommandSet)
{
#if (GEN_NOR == TRUE) /* GenNOR Flash Module Option */
    case CMDSET_01:
    case CMDSET_03:
        status = GenNORFlash_InitDevice(
            (UINT32) flash_start_add,
            &device_info,
            &MTD_Devices[list_loop]);
        cmdset01cnt++;

```

```

        break;
    #endif /* GEN_NOR == TRUE */

    #if (M18_NOR == TRUE)
        case CMDSET_0200:
            status = M18Flash_InitDevice(
                (UINT32) flash_start_add,
                &device_info,
                &MTD_Devices[list_loop]);
            cmdset0200cnt++;
            break;
    #endif /* M18_NOR == TRUE */
    #if (SecondSource_NOR == TRUE)
        case SS_CMDSET_XX
            status = SecondSourceFlash_InitDevice( (UINT32)
flash_start_add,
                                                    &device_info,
                                                    &MTD_Devices[list_loop]);
    #endif
        default:
            status = FLASH_ERR_UNSUPPORTED_DEVICE;
            break;
    }
}

```

## Creating the SecondSourceFlash\_InitDevice Function

A new function must be created.

**Note:** In the following code sample, text in *italics* indicates information you must define.

```

/*****
*
* SecondSourceFlash_InitDevice
*
* DESCRIPTION:
* Initializes a FlashDevice Structure.
*
* PARAMETERS:
* ret_device_ptr - [OUT] contains a pointer to the FlashDevice
*                 structure that holds information about the

```



```
*      device(s)
*
* RETURNS:
* If insufficient memory is available for the function,
FLASH_ERR_MALLOC
* is returned. If the function is unable to create a semaphore,
* FLASH_ERR_SEMAPHORE is returned. Otherwise, FLASH_ERR_NONE is
returned.
*
*****/

FLASH_ERROR SecondSourceFlash_InitDevice( UINT32address,
                                           const FLASH_DeviceInfo*device_info_ptr,
                                           struct FlashDevice**ret_device_ptr)
{
    struct FlashDevice *device;

    /* Allocate the memory for the FlashDevice structure */
    device = (struct FlashDevice *)mOS_Malloc(sizeof(struct Flash-
Device));
    if (device == NULL)
    {
        return FLASH_ERR_MALLOC;
    }

    /* flash layout */
    /* hardcoding the configuration for a single x16 flash device
on the bus.
    * See below for additional configuration possibilities */
    device->Layout.BusWidth = BUS_WIDTH_16BIT;
    device->Layout.Interleave =
(UINT8) (POWER_2(FLASH_ConfigGroupingSingle));

    /* standard device information */
    device->StartingAddress = address;

    /* Set to the total device size in bytes, example 128Mb device
set to =0x1000000*/
    device->Size = ?
    device->SpareSize = 0;
```

```
    device->Features = (UINT32) device_info_ptr->DeviceCapabili-
ties; /* ??*/

/* Set to the maximum buffer size of the device or 1024, which-
ever is smaller.*/

    device->BufferSize = ?

/* control/object mode sizes are zero if Second Source vendor
does not have write restrictions like Micron M18 device. */

device->ControlModeValidSize    = 0;
device->ControlModeInvalidSize = 0;
device->ObjectModeSize         = 0;
device->FlashType               = SUBARRAY_TYPE_NOR;

/* partition information */

/* The Partition structure is used by the Low level to enable
Hardware Read While Write or Software Read While Write. The Par-
tition structure will maintain the state of the partition and the
partitions ability for HWRWW or SWRWW (set later in Init) based
on the settings for the location of the data volume and code vol-
ume.

/* The following is set for a multiple partition device, assumes
all partitions are uniform in size*/

    device_info_ptr->PartitionRegionCount = 1;
    device_info_ptr->PartitionRegionInfo[0].PartitionCount = 16;
    device_info_ptr->PartitionSize =
    device->Size / device_info_ptr->PartitionRegionInfo[0].Parti-
tionCount;

/* this is the Device size as 2^n. for a 128Mb device n=0x18; */
/* this value can be pulled dynamically from the Device Geometry
field in CFI.*/

    device_info_ptr->DeviceSizeExponent = 0x18;
    device->Partitions = CreatePartitions_v1_4(
        &device->Layout,
        device_info_ptr,
        &device->NumPartitions);

if (device->Partitions == NULL)
{
    mOS_Free(device);
    device = NULL;
}
```

```
        return FLASH_ERR_MALLOC;
    }

    /* erase region information */
    /* The erase region is used to describe the block size layout in
    flash. This structure was pulled directly from the CFI Device
    Geometry. Following is an example of how to hardcode the informa-
    tion. This could be set dynamically by reading the CFI informa-
    tion on the Erase block Region information */

    device_info_ptr->EraseRegionCount = 1;

    /* set to the size of the block in bytes that will contain the
    FDI volume.*/
    device_info_ptr->EraseBlockRegionInfo[0].BlockSize = 0x20000;
    device_info_ptr->EraseBlockRegionInfo[0].BlockCount =
        device->Size / device_info_ptr->EraseBlockRegion-
        Info[0].BlockSize;

    device->NumEraseRegions = device_info_ptr->EraseRegionCount;
    device->EraseRegions = CreateEraseRegions(
        &device->Layout,
        device_info_ptr->EraseBlockRegionInfo,
        device_info_ptr->EraseRegionCount,
        &device->TotalNumBlocks);

    if (device->EraseRegions == NULL)
    {
        mOS_Free(device->Partitions);
        device->Partitions = NULL;
        mOS_Free(device);
        device = NULL;
        return FLASH_ERR_MALLOC;
    }

    /* Write State Machine Semaphore */
    if (mOS_MutexCreate(&device->WsmSemaphore, FALSE, NULL) !=
    OS_SemSuccess )
    {
        mOS_Free(device->EraseRegions);
        device->EraseRegions = NULL;
        mOS_Free(device->Partitions);
        device->Partitions = NULL;
    }
}
```

```
    mOS_Free(device);
    device = NULL;
    return FLASH_ERR_SEMAPHORE;
}
/* allocate buffer for buffer fills */
device->BufferPtr = (UINT8 *) mOS_Malloc(device->BufferSize);
if (device->BufferPtr == NULL)
{
    return FLASH_ERR_MALLOC;
}
/* setup common function pointers */
device->Read = SecondSourceFlash_Read;
device->Write = SecondSourceFlash_Write;
device->BlankCheck = SecondSourceFlash_BlankCheck;
device->Erase = SecondSourceFlash_Erase;

/* setup block lock function pointers, if supported */
if (SS_block_locking_individual) /*Create a #define to state
if SS will have individual Block locking */
{
    device->Lock = SecondSourceFlash_Lock;
    device->Lockdown = SecondSourceFlash_Lockdown;
    device->Unlock = SecondSourceFlash_Unlock;
    device->ReadLockStatus =
SecondSourceFlash_ReadLockStatus;
}
else
{
    device->Lock = NULL;
    device->Lockdown = NULL;
    device->Unlock = NULL;
    device->ReadLockStatus = NULL;
}

/* device structure initialized, return success */
*ret_device_ptr = device;

return FLASH_ERR_NONE;
```

```
}

```

Within the device structure there is a layout structure that is used by low-level macros to send the commands to the Flash in the proper manner, depending on how the many chips are on the bus. For a typical system there will be a single x16 Flash device on the bus. This configuration has been set in the previous example.

The code to dynamically set the layout can be seen in `mtd_cfir.c` at the top of the `CFI_ReadtoBuffer` function.

## FlashLayout

See “Structures and Defines” on page 20 for structure definitions.

Interleave is set as a power of 2 raised by the corresponding grouping define in your system.

```
#define FLASH_ConfigGroupingSingle 0x00
#define FLASH_ConfigGroupingPaired 0x01
#define FLASH_ConfigGroupingQuad 0x02
#define FLASH_ConfigGroupingOctal 0x03

```

BusWidth is set to one of the following defines that corresponds to the bus width for the Flash in your system.

```
#define BUS_WIDTH_8BIT 1
#define BUS_WIDTH_16BIT 2
#define BUS_WIDTH_32BIT 4

```

## Runtime Selection of the Second Source Vendor

The method described in “Compile Option Method” on page 3 illustrates how to switch between a Micron Flash device and a second source device through a compile option. To allow the selection between a Micron Flash device and the Second Source Flash with a single compile, the following changes must be made to “Compile Option Method” on page 3.

A new function must be created that pulls the manufacture ID from the second source. Then, the compile option is replaced with an *if else* statement that determines the correct code path, depending on the manufacture ID.

The following code is located in the `CreateDevices` function in `MTD_INIT.c`:

```
if (current_element->OEMSubArrayPtr->Type == SUBARRAY_TYPE_NOR)
{
    /* Need to create a new function that will read the manufacture
    ID
    * for the Second Source flash device */
    If ( GetSSManID(FlashQueryAddress) == SS_Man_ID )
    {
        device_info. PrimaryCommandSet = SS_CMDSET_XX
    }
    else

```

```
{
    status = MTD_GetFlashDeviceInfo(&device_info,
                                    (VOID_PTR)FlashQueryAddress,
                                    (VOID_PTR)FlashQueryAddress,
                                    buswidth);

    if (status != FLASH_ERR_NONE)
    {
        break;
    }

    num_dev_on_bus = (UINT8)
        (1<< (device_info.Configuration & FLASH_ConfigGroupingMask));

    /* check to make sure whole device has been given.
     * does device size not less the total length in the linked
     list.
     */
    current_flash_length =0;
    for (test_token = current_element; test_token != NULL;
         test_token = test_token->Next)
    {
        if( test_token->OEMSubArrayPtr == NULL )
        {
            continue;
        }
        current_flash_length += test_token->OEMSubArrayPtr->SubAr-
rayLength;
    }
    if (current_flash_length !=
        (UINT32) (1<< (device_info.DeviceSizeExponent +
                     (num_dev_on_bus>>1))))
    {
        status = FLASH_ERR_DEVICE_SIZE_MISMATCH;
        break;
    }
}
```

## Precondition Functions

This section describes the functions that must be created to replace the low-level functions. When creating the functions, replace “SecondSource” in the function names described in this section with the name of the Flash memory device being used. For example, name the function labeled “SecondSourceFlash\_Read” in this section to “M18Flash\_Read” when porting the low-level driver for Micron® StrataFlash® Cellular Memory (M18).

Precondition functions are not relocated. These functions are listed in Table 1 and described in this section. These functions are used when data is stored in a Flash memory device or Flash partition separate from the executable code. The structures for these functions are shown and explained in “Structures and Defines” on page 20.

**Table 1: Low-Level Driver Precondition Functions**

Function Name	Description
<i>SecondSourceFlash_Read</i>	Reads information from the Flash memory device.
<i>SecondSourceFlash_Write</i>	Writes information to the Flash memory device.
<i>SecondSourceFlash_Erase</i>	Erases information from the Flash memory device.
<i>SecondSourceFlash_Lock</i>	Locks a specified Flash memory device.
<i>SecondSourceFlash_Unlock</i>	Unlocks a specified Flash memory device.

Once the following functions are implemented, you will have a functional second source low-level driver. If you want to add additional functionality, refer to the section “Compile Option Method” on page 3.

## Multithreading Synchronization

The FDI core allows for multithread access to the MTD. The multithread synchronization is controlled in the SSFlash\_x functions with the write state machine semaphore (WsmSemaphore). At the beginning of each SSFlash\_x function, the semaphore must be taken and then must be released before exiting the function.

```
mOS_MutexPend(&device->WsmSemaphore, OS_SemWaitForever);
mOS_MutexRelease(&device->WsmSemaphore);
```

The WSMSemaphore is sufficient synchronization if the FDI volume is contained in its own hardware partition in the Flash Device. This will not protect from faults if code is within the same hardware partition as FDI.

## Software READ-While-WRITE Synchronization

If code and the FDI volume share the same hardware partition, the code must prevent code accesses while the device is performing a task other than READ, which puts the device partition in a state other than read array. This is accomplished by controlling the interrupts in the low-level driver. To enable software READ-While-WRITE, call disable interrupt functions after the pending of WsmSemaphore at the entry of the functions. The prior to the exit call, enable interrupts before the release of the WsmSemaphore.

```
System.DisableInterrupts();
System.EnableInterrupts();
```

**Note:** The source file containing the precondition functions must be relocated into RAM in the scatter file for the linker.

## SecondSourceFlash\_Read

Reads data from the Flash memory device

### Syntax

```
FLASH_ERROR SecondSourceFlash_Read (
    struct FlashDevice *device,
    UINT32             offset,
    UINT32             len,
    UINT8              *buffer,
    UINT32             *retlen )
    const void         *part_vars);
```

### Parameters

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. See "Structures and Defines" on page 20 for more information.
offset	(IN) The offset in bytes from the first Flash memory address to the location to send the Flash memory commands.
len	(IN) Amount in bytes to read from the Flash memory device.
buffer	(IN) Pointer to the location to store the data read from the Flash memory device.
retlen	(OUT) The number of bytes read from the Flash memory device.
part_vars	(IN) Pointer to the structure with additional variables from the higher-level API.

### Error Codes/Return Values

Error Code	Description
FLASH_ERR_OUT_OF_BOUNDS	The area being read does not lie within the Flash device.
FLASH_ERR_NONE	The read was successful.

### Description

The *SecondSourceFlash\_Read* function performs the following tasks:

- Erases a single Flash block and returns.
- Verifies the offset is within the Flash memory device.
- Verifies that the offset is translated into the physical address to the Flash memory device.

## SecondSourceFlash\_Write

Writes information to the Flash memory device

### Syntax

```
FLASH_ERROR SecondSourceFlash_Write(
    struct FlashDevice *device,
    UINT32             offset,
    UINT32             len,
    const UINT8        *buffer,
    UINT32             *retlen,
    const void         *part_vars);
```

### Parameters

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. See "Structures and Defines" on page 20 for more information.
offset	(IN) The offset in bytes from the Flash memory device start address to the location to send the Flash memory commands.
len	(IN) Amount in bytes to write from the Flash memory device.
buffer	(IN) Pointer to the data to write to the Flash memory device.
retlen	(OUT) The number of bytes written to the Flash memory device.
part_vars	(IN) Pointer to the structure with additional variables from the higher-level API.

### Error Codes/Return Values

Error Code	Description
FLASH_ERR_NONE	Success
FLASH_ERR_OUT_OF_BOUNDS	Offset is greater than the device size indicated in the FlashDevice structure.
FLASH_ERR_MALLOC	The function is unable to allocate memory.
FLASH_ERR_INVALID_SEQUENCE	An invalid command sequence was issued to the Flash device.
FLASH_ERR_INVALID_ADDRESS	Offset does not refer to a Flash memory block start location.
FLASH_ERR_SEQUENCE	Flash Device Status register indicated a sequence error.
FLASH_ERR_VPP	Flash device status register indicated a V <sub>pp</sub> error, which indicates that the supply voltage to the Flash device was not within the proper range.
FLASH_ERR_LOCKED	Flash device status register indicated the block is locked.
FLASH_ERR_WRITE	Flash device status register indicated an error during a write.

## Description

This must be implemented to align data writes on bus and program buffer boundaries. Because calling functions can send data to odd Flash byte addresses, you must align the data writes to the buffer. Utilize the alignment buffer in the FlashDevice structure member BufferPtr. Refer to M18Flash\_Write for a sample alignment method.

The *SecondSourceFlash\_Write* function performs the following tasks:

- Aligns data writes on bus and program buffer boundaries.
- Performs writes larger in size than the program buffer. Writes can be up to, but not exceed the size of the Flash block. For performance, it is recommended that you release the WsmSemaphore and interrupts (if software READ-While-WRITE) between each buffer write or specific time interval for the system.
- Verifies the offset is within the Flash memory device.
- Translates the offset into the physical address to the Flash memory device.
- Sends commands to the Flash memory device and calls low-level relocated or non-relocated function, according to the partition type specified in the offset.
- Acquires the write state machine semaphore in the FlashDevice structure before calling low-level write routine and releases the semaphore after calling low-level write routine.
- Translates the control function status return (Flash memory status register value) into a valid FLASH\_Error.
- If the device supports individual block lock and unlocking, add a call to Flash\_Unlock at the entry of the function and then Flash\_Lock at the exit. This increases security and decreases the likelihood of an accidental write.

## SecondSourceFlash\_Erase

Erases information from the Flash memory device

## Syntax

```
FLASH_ERROR SecondSourceFlash_Erase (
    struct FlashDevice *device,
    UINT32                offset);
```

## Parameters

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. See "Structures and Defines" on page 20 for more information.
offset	(IN) The offset (in bytes) from the Flash memory device start address to the location to send the Flash memory commands.

## Error Codes/Return Values

Error Code	Description
FLASH_ERR_NONE	Success
FLASH_ERR_OUT_OF_BOUNDS	Offset is greater than the device size indicated in the FlashDevice structure.
FLASH_ERR_INVALID_ADDRESS	Offset does not refer to a Flash memory block start location.
FLASH_ERR_INVALID_SEQUENCE	Flash Device Status register indicated a sequence error.
FLASH_ERR_VPP	Flash Device Status register indicated a V <sub>pp</sub> error.
FLASH_ERR_LOCKED	Flash Device Status register indicated the block is locked.
FLASH_ERR_ERASE	Flash Device Status register indicated an error during erase.

## Description

The *SecondSourceFlash\_Erase* function performs the following tasks:

- Verifies the offset is within the Flash memory device.
- Translates the offset into the physical address to the Flash memory device.
- Specifies the offset for the partition type to call relocated or non-relocated low-level functions to send the commands to the Flash memory device.
- Acquires the Write State machine semaphore in the FlashDevice structure before calling low-level write routine and releases the semaphore after calling low-level write routine.
- If the device supports individual block lock and unlocking, add a call to *Flash\_Unlock* at the entry of the function and then *Flash\_Lock* at the exit. This increases security and decreases the likelihood of an accidental write.

## SecondSourceFlash\_Lock

Locks a block within the Flash memory device

## Syntax

```
FLASH_ERROR SecondSourceFlash_Lock (
    struct FlashDevice *device,
    UINT32                offset);
```

## Parameters

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. See "Structures and Defines" on page 20 for more information.
offset	(IN) The offset (in bytes) from the Flash memory device start address to the location to send the Flash memory commands.

## Error Codes/Return Values

Error Code	Description
FLASH_ERR_NONE	Success
FLASH_ERR_OUT_OF_BOUNDS	Offset is greater than the device size indicated in the FlashDevice structure.
FLASH_ERR_INVALID_ADDRESS	Offset does not refer to a Flash memory block start location.

## Description

The *SecondSourceFlash\_Lock* function performs the following tasks:

- Verifies the offset is within the Flash memory device.
- Translates the offset into the physical address to the Flash memory device.
- Calls the relocated low-level function to send the commands to the Flash memory device.

## SecondSourceFlash\_Lockdown

Locks down a block within the Flash memory device

## Syntax

```
FLASH_ERROR SecondSourceFlash_Lockdown (
    struct FlashDevice *device,
    UINT32 offset);
```

## Parameters

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. (See "Structures and Defines" on page 20.)
offset	(IN) The offset (in bytes) from the Flash memory device start address to the location to send the Flash memory commands

## Error Codes/Return Values

Error Code	Description
FLASH_ERR_NONE	Success
FLASH_ERR_OUT_OF_BOUNDS	Offset is greater than the device size indicated in the FlashDevice structure.
FLASH_ERR_INVALID_ADDRESS	Offset does not refer to a Flash memory block start location

## Description

The *SecondSourceFlash\_Lockdown* function performs the following tasks:

- Verifies the offset is within the Flash memory device.
- Translates the offset into the physical address to the Flash memory device.
- Calls the relocated low-level function to send the commands to the Flash memory device.

## **SecondSourceFlash\_Unlock**

Unlocks a block within a Flash memory device

### **Syntax**

```
FLASH_ERRORSecondSourceFlash_Unlock (
    struct FlashDevice *device,
    UINT32                offset);
```

### **Parameters**

Parameter	Description
device	(IN) Pointer to the FlashDevice structure associated with the Flash memory device to operate. (See "Structures and Defines" on page 20.)
offset	(IN) The offset (in bytes) from the Flash memory device start address to the location to send the Flash memory commands.

### **Error Codes/Return Values**

Error Code	Description
FLASH_ERR_NONE	Success
FLASH_ERR_OUT_OF_BOUNDS	Offset is greater than the device size indicated in the FlashDevice structure.
FLASH_ERR_INVALID_ADDRESS	Offset does not refer to a Flash memory block start location

### **Description**

The *SecondSourceFlash\_Unlock* function performs the following tasks:

- Verifies the offset is within the Flash memory device.
- Translates the offset into the physical address to the Flash memory device.
- Calls the relocated low-level function to send the commands to the Flash memory device.

## Replacing Low-level Flash Memory Device Functions

The functions in this section must be supplied by your low-level driver. These functions must also adhere to Micron FDI file system error handling.

The file system accepts a successful return value of 0 from the low-level driver and labels it FLASH\_ERR\_NONE. Values other than 0 are reported through the file system with an appropriate descriptive error code. The supported descriptive hardware error codes are in the MTD\_ERR.H file. Table 2 describes all error codes that the memory technology device (MTD) might return.

**Table 2: Low-Level Error Codes**

#	Errno Name	Meaning
0x00	FLASH_ERR_NONE	The operation completed successfully.
0x01	FLASH_ERR_VPP	V <sub>PP</sub> was not within acceptable limits during a PROGRAM or ERASE operation.
0x02	FLASH_ERR_SEQUENCE	An invalid command sequence was sent to the Flash memory device. This error code applies only to multicycle command sequences.
0x03	FLASH_ERR_LOCKED	A PROGRAM or ERASE operation was attempted on a locked block.
0x04	FLASH_ERR_ERASE	An ERASE operation failed.
0x05	FLASH_ERR_WRITE	A PROGRAM operation failed.
0x06	FLASH_ERR_IO	A general I/O failure occurred.
0x10	FLASH_ERR_NOT_SUPPORTED	The Flash memory device does not support the requested feature.
0x11	FLASH_ERR_OUT_OF_BOUNDS	The requested operation is attempting to act on an address that does not lie within the Flash memory device address range.
0x12	FLASH_ERR_ARRAY_TYPE	An attempt to perform an operation was made on the incorrect Flash memory array type.
0x13	FLASH_ERR_INVALID_PARTITION	An attempt to program protection registers was made using a Flash memory address that is outside the first partition.
0x14	FLASH_ERR_MALLOC	A call to mOS_Malloc failed.
0x15	FLASH_ERR_MAX_ARRAYS	An attempt to add another Flash memory array was made when the limit had already been reached.
0x16	FLASH_ERR_INVALID_ADDRESS	An address other than a block address was used on an operation that requires a block address.
0x17	FLASH_ERR_INCOMPATIBLE_DEVICES	The devices used are not compatible with one another.
0x18	FLASH_ERR_SEMAPHORE	An error occurred while performing an operation on internal semaphores.
0x19	FLASH_ERR_CONFIG_UNSUPPORTED	The Flash memory configuration is neither a single 16-bit Flash memory device on a 16-bit data bus, nor two 16-bit Flash devices on a 32-bit data bus.
0x1A	FLASH_ERR_UNSUPPORTED_DEVICE	The MTD does not support the Flash memory device being used.
0x1B	FLASH_ERR_UNKNOWN_CFI_VERSION	A Flash memory device is using an extended CFI table that the MTD does not understand.
0x1C	FLASH_ERR_SUBARRAY_DEF	An error occurred while gathering information about the OEM sub array layout.
0x1D	FLASH_ERR_INTC_INIT	An error occurred while obtaining function pointers to the interrupt control functions.

**Table 2: Low-Level Error Codes**

#	Errno Name	Meaning
0x1E	FLASH_ERR_MAX_DEVICES	An attempt to add another Flash memory device was made when the limit had already been reached.
0x1F	FLASH_ERR_INVALID_OFFSET	An invalid offset was used for an operation that requires a specific offset.
0x20	FLASH_ERR_MEMORY_MAP_OVERLAP	The OEM has specified its sub array configuration so that two or more sub arrays overlap in the memory map.
0x21	FLASH_ERR_NON_CONTIGUOUS_ADDR	The OEM has specified its sub array configuration so that a gap occurs in the memory map within a single Flash memory device.
0x22	FLASH_ERR_DEVICE_SIZE_MISMATCH	The managed area size specified in the OEM sub arrays does not match the Flash memory device size.

## Structures and Defines

The following FlashDevice structure contains information about the Flash memory device. Table 3 on page 21 defines the structure members.

```

struct FlashDevice
{
    UINT32          StartingAddress;
    UINT32          Size;
    UINT32          BufferSize;
    UINT32          Features;
    struct FlashLayout Layout;
    UINT32          ControlModeValidSize;
    UINT32          ControlModeInvalidSize;
    UINT32          ObjectModeSize;
    UINT8           NumPartitons;
    struct FlashPartition *Partitions;
    UINT8           NumEraseRegions;
    struct EraseRegion *EraseRegions;
    OS_Semaphore    WsmSemaphore;
    FLASH_ERROR     (*Read)(struct FlashDevice *,
                           UINT32, UINT32, UINT8 *, UINT32 *);
    FLASH_ERROR     (*Write)(struct FlashDevice *, UINT32,
                             UINT32, const UINT8 *, UINT32 *);
    FLASH_ERROR     (*Erase)(struct FlashDevice *,
                              UINT32);
    FLASH_ERROR     (*Lock)(struct FlashDevice *, UINT32);
    FLASH_ERROR     (*Lockdown)(struct FlashDevice *,
                                 UINT32);
    FLASH_ERROR     (*Lock)(struct FlashDevice *, UINT32);
}

```

```

FLASH_ERROR      (*ReadLockStatus) (struct FlashDevice
                    *, UINT32, UINT32 *);

FLASH_ERROR      (*ReadProtReg) (struct FlashDevice *,
                    UINT32, UINT32, UINT8 *, UINT32 *);

FLASH_ERROR      (*WriteProtReg) (struct FlashDevice *,
                    UINT32, UINT32, const UINT8 *, UINT32
                    *);

void             *Private;
};

```

**Table 3: FlashDevice Structure Members**

Structure Member	Description
StartingAddress	The physical starting address of the Flash memory device. Value is pulled from defines in PlatDefineArrayParameter( ).
Size	The Flash memory device size. If multiple Flash devices are in parallel, this value indicates the total Flash device size.
BufferSize	The write buffer size in the Flash memory device. If multiple Flash devices are in parallel, this value indicates the total write buffer size.
Features	Holds the list of all the features in the Flash memory device. The following defines are ORed for the set the available features: <pre> #define FLASH_CapsCFIEnabled          0x01 #define FLASH_CapsProgSusRead        0x04 #define FLASH_CapsEraseSusRead       0x08 #define FLASH_CapsEraseSusProg       0x10 #define FLASH_CapsFullChipErase      0x20 #define FLASH_CapsQueuedErases       0x40 #define FLASH_CapsInstIndvBlockLocking 0x80 </pre>
Layout	Pointer to a FlashLayout structure. “Structures and Defines” on page 20 for more information
ControlModeValidSize	For next-generation multilevel cell (MLC) devices. Size of the area that can be written multiple times.
ControlModeInvalidSize	For next-generation MLC devices. Write restricted size for multiple write area.
ObjectModeSize	For next-generation MLC devices. Size of the area that can be written only once.
NumPartitions	Number of hardware partitions in the Flash memory device.
Partitions	Pointer to an array of FlashPartition structures. The number of partitions in the Flash memory device determines the array size. See “Structures and Defines” on page 20 for more information.
NumEraseRegions	Number of regions within the Flash memory device with different size erase block. For example, Flash memory devices with a parameter region that has two erase regions.
EraseRegions	Pointer to an array of EraseRegions structure. The number of erase regions determines the array size. See “Structures and Defines” on page 20 for more information.
WsmSemaphore	Write state machine semaphore. Only one PROGRAM/ERASE operation can occur at any moment (cannot suspend ERASE to WRITE). However, another process/task/thread can read the Flash memory device during PROGRAM/ERASE operations (programming or erasing can be suspended to READ.)

The FlashDevice structure contains an EraseRegion structure, which lists structures containing information about erase regions within the Flash memory device. The list is created at initialization and is based on concatenated sub arrays. If the block sizes are

different within the sub array, then create a new concatenated erase region. Within each EraseRegion structure the byte offset into the erase region, the block size, and the number of blocks are tracked. The list of arrays requires at least one Flash memory device containing at least one erase region.

The following example shows the EraseRegion structure:

```

struct EraseRegion
{
    UINT32          Offset;
    UINT32          BlockSize;
    UINT32          BlockCount;
};
    
```

**Table 4: EraseRegion Structure Members**

Structure Member	Description
Offset	Offset in bytes from the Flash memory device start address.
BlockSize	Block size in the region.
BlockCount	Number of blocks in the region.

The FlashPartition structure contains information about hardware partitions in the Flash memory device. Each Flash device uses one global FlashDevice structure. Within the FlashDevice structure is an array of FlashPartition structures. The number of hardware partitions in the Flash device determines the number of structures in the array.

### Example

A 128-bit L18 device contains 16 hardware partitions and a 128-bit K18 device contains one hardware partition.

The partition structure determines whether the partition region includes any executable code. If it does include executable code, a relocated WRITE or ERASE routine must be called. Also, the FlashPartition structure holds the hardware partition state if hardware READ-While-WRITE is used. As a result, a READ operation can suspend a WRITE to READ the data from the Flash memory device.

The following example shows the FlashPartition structure:

```

struct FlashPartition
{
    UINT32          Offset;
    UINT32          Size;
    RwwType         Type;
    PartitionState  State;
};
    
```

**Table 5: FlashPartition Structure Members**

Structure Member	Description
Offset	Offset from the beginning of the Flash memory device to the partition location.
Size	Hardware partition size in the device.
Type	RWW_TYPE_SW for software READ-While-WRITE. RWW_TYPE_HW for hardware READ-While-WRITE. See "Structures and Defines" on page 20 for more information.
State	See "Structures and Defines" on page 20.

A hardware partition can be in one of several read states. The PartitionState enumeration contains all read states that a Flash memory partition can be in. The RwwType enumeration tells the driver the type of information to read.

The following example shows the PartitionState typedef enum:

```
typedef enum
{
    STATE_READ_ARRAY,
    STATE_READ_STATUS,
    STATE_READ_ID,
    STATE_READ_QUERY,
}
PartitionState;
```

The following example shows the RwwType typedef enum:

```
typedef enum
{
    RWW_TYPE_SW,
    RWW_TYPE_HW,
}
RwwType;
```

**Table 6: RwwType Typedef Enum**

Structure Member	Description
RWW_TYPE_SW	RWW_TYPE_SW for software READ-While-WRITE.
RWW_TYPE_HW	RWW_TYPE_HW for hardware READ-While-WRITE.

The following FlashLayout structure details the layout of particular Flash memory device. A device can be one or more physical Flash devices interleaved across the data bus.

```

struct FlashLayout
{
    UINT8          BusWidth;
    UINT8          Interleave;
};
    
```

**Table 7: FlashLayout Structure Members**

Structure Member	Description
BusWidth	The data bus width in bytes. The member value is one of the two following #defines: #define BUS_WIDTH_16BIT #define BUS_WIDTH_32BIT
Interleave	The number of devices across the data bus. For example, a single device x16 bus interleave is equal to 1.

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900  
[www.micron.com/productsupport](http://www.micron.com/productsupport) Customer Comment Line: 800-932-4992

Micron, the Micron logo, and StrataFlash are trademarks of Micron Technology, Inc. All other trademarks are the property of their respective owners.