Technical Note

Software Driver for M29EW NOR Flash Memory

Introduction

This technical note describes the C library source code for the M29EW parallel NOR Flash memory device using the Flash device driver software interface V3. The M29EW_32Mb-2Gb_device_driver.c and M29EW_32Mb-2Gb_device_driver.h files contain libraries for accessing the M29EW Flash memory device.

Also included here are programming model and device operation overviews and a summary of the Flash device driver interface. A complete description is available from your Micron distributor.

The accompanying platform-independent source code has a small footprint, which can be applied to any target hardware. Minimal code changes are required to compile and run, plus extensive code comments enable easy modifications for the individual target. This software has been tested on a target platform and is usable in C and C++ environments.

The device driver interface enables changeable configurations, compiler-independent data types, and a unique access mode for a broad range of Flash memory devices. Applications supporting the device driver standard can implement any Flash device with the same interface without changes to the code. Recompiling with a new software driver is all that is required to control a new device.

This technical note does not replace the M29EW data sheet, and a copy of it is required to follow extensive references to data sheet information.

M29EW Programming Model

The M29EW device (128Mb, 256Mb, 512Mb, 1Gb, and 2Gb) is a nonvolatile Flash memory that can be electrically erased at the block level and programmed in-system through special coded command sequences on most standard microprocessor buses.

The memory array is divided into 64-Kword/128KB uniform blocks that can be erased independently, making it is possible to preserve valid data while old data is erased.

The M29EW device has one extra 128-word block (extended memory block) that is only accessible using a dedicated command. The extended block can be protected, which is useful for storing security information. However, the protection is not reversible, which means that once the extended block is protected, the protection cannot be undone.

Each block can be erased separately. PROGRAM and ERASE can be suspended to either read from or program to any other block and then be resumed. Each block can be programmed and erased over 100,000 cycles.

The blocks can be protected to prevent an accidental program or erase from modifying the memory. PROGRAM and ERASE commands are written to the memory's command interface. An on-chip PROGRAM/ERASE controller (P/E.C.) handles timings necessary for PROGRAM and ERASE operations. The end of a PROGRAM or ERASE operation can be detected and any error conditions identified. The command set required to control the memory is consistent with JEDEC standards.

Bus Operations and Commands

Most M29EW features are available by using two standard bus operations: READ and WRITE. READ operations retrieve data or status information from the device. WRITE operations are interpreted by the device as commands to modify either the data stored or the device's behavior. The M29EW device recognizes only certain special WRITE operation sequences as commands, and these various commands are listed in the data sheet Command Tables. The main commands can be classified as follows:

- READ/RESET
- AUTO SELECT
- BLOCK/CHIP ERASE
- PROGRAM
- FAST PROGRAM (a set of FAST PROGRAM commands: WRITE TO BUFFER PROGRAM and UNLOCK BYPASS)
- PROGRAM/ERASE SUSPEND and RESUME
- READ COMMON FLASH INTERFACE QUERY
- CHIP UNPROTECT
- VOLATILE BLOCK PROTECT
- NON VOLATILE BLOCK PROTECT
- PASSWORD BLOCK PROTECT

The READ/RESET command returns the M29EW to read mode where it behaves as a ROM. In this mode, a READ operation outputs the data stored at the specified device address onto the data bus.

The AUTO SELECT command puts the device in auto select mode. Once in this mode, the system can read the manufacturer and device codes, the protection status of each block (block protection status), and the extended memory block protection indicator.

Three consecutive BUS WRITE operations are required to issue the AUTO SELECT command.

The PROGRAM command is used to modify the data stored at the specified device address. Note that programming can only change bits from 1 to 0. If an attempt is made to change a bit from 0 to 1 using the PROGRAM command, the command will be executed and no error will be returned. However, the bit will remain unchanged. Therefore, it may be necessary to erase the block before programming to addresses within it. The command requires four BUS WRITE operations. The final WRITE operation latches the address and data in the internal state machine and starts the PROGRAM/ERASE controller.

The BLOCK ERASE command can be used to erase a list of one or more blocks. It sets all bits in the unprotected selected blocks to 1. All previous data in the selected blocks is lost. Six BUS WRITE operations are required to select the first block in the list. Each additional block in the list can be selected by repeating the sixth BUS WRITE operation using the address of the additional block.

The CHIP ERASE command is used to erase the entire memory. Six BUS WRITE operations are required to issue the CHIP ERASE command and start the PROGRAM/ERASE controller. If a block is protected, it is ignored and all the other blocks are erased. If all blocks are protected, the CHIP ERASE operation appears to start but will terminate within about 100 seconds, leaving the data unchanged. No error condition is given when protected blocks are ignored.

Issuing the PROGRAM/ERASE SUSPEND command during a PROGRAM or ERASE operation will temporarily place the M29EW in program/erase suspend mode. While an ERASE operation is suspended, the blocks not being erased can be read or programmed as if in the reset state of the device. While a PROGRAM operation to one block is being suspended, the rest of the device can be read. This allows immediate access to information stored in the M29EW device without having to wait for the PROGRAM or ERASE operation to complete. The PROGRAM or ERASE operation is resumed when the device receives the PROGRAM/ERASE RESUME command.

The M29EW has a set of FAST PROGRAM commands that improve programming throughput:

- WRITE TO BUFFER PROGRAM command uses the device's 512-word/1024-byte write buffer to speed up programming.
- UNLOCK BYPASS command is used to place the device in unlock bypass mode. When the device enters unlock bypass mode, the two initial unlock cycles required in the standard PROGRAM command sequence are no longer needed. Only two write cycles instead of the normal four cycles are required to program data, providing a faster total programming time.

The READ CFI QUERY command reads data from the common flash interface (CFI), which is used to identify the number of blocks in the device and the block addresses. The interface also contains information relating to the typical and maximum PROGRAM and ERASE times. This allows the implementation of software timeouts and prevents waiting indefinitely for a defective Flash memory device to finish programming or erasing. For further information about the CFI, please refer to the CFI specification available from www.jedec.org or from your Micron distributor.

The device is shipped with all blocks unprotected. The block protection status can be read, for example, by performing a read electronic signature. The M29EW has three dif-

ferent software protection modes: volatile protection, nonvolatile protection, and password protection. On first use, all parts default to nonvolatile protection mode operate and the customer is free to activate the nonvolatile or the password protection mode. A protection mode is activated by setting a bit of the lock register. For details, refer to the data sheet.

Status Register

The M29EW discrete device has one status register, which provides information on the current or previously executed PROGRAM or ERASE operation. The various bits convey information and errors on the operation. BUS READ operations from any address always read the status register during PROGRAM and ERASE operations. The status register is also read during an ERASE SUSPEND when accessing an address within a block being erased.

A Detailed Example

The commands tables provided in the M29EW data sheet describe the WRITE operation sequences recognized as valid commands by the PROGRAM/ERASE controller. As an example, consider the following example of programming the value 9465h to the address 03E2h. The following C sequence is required:

```
*(NMX_uint16*)(0xAAA) = 0xAA;    /* 1st cycle:    (Programming Command Sequence) */
*(NMX_uint16*)(0x555) = 0x55;    /* 2nd cycle:    (Programming Command Sequence) */
*(NMX_uint16*)(0xAAA) = 0xA0;    /* 3rd cycle:    (Programming Command Sequence) */
*(NMX_uint16*)(0x3E2) = 0x9465;  /* final cycle:  (Program Addr and Value) */
```

Where uword is defined as the following 16-bit value:

```
typedef unsigned short uword
```

The first three addresses and values shown in this example are used only to initiate the PROGRAM command. This example assumes that the address 0000h in the M29EW device is mapped to the address 0000h in the microprocessor address space. In practice, the Flash memory device is likely to have a base offset that must be added to the address.

While the device is programming to the specified address, READ operations on the involved block can still access the status register bits. Status register bits DQ7 or DQ6 can be used to determine whether the operation has completed.

Using the Software Driver

General Considerations

The software drivers described in this application note are intended to simplify the process of developing application code in C for the M29EW Flash memory device.

This software driver supports the Flash device driver interface implemented in all software drivers. As a result, future device changes will not necessarily lead to code changes in application environments.

Note: To meet compatibility requirements, the Flash device driver interface allocates numbers to each block in a Flash memory device, starting from 0 (block 0 always has address offset 0) and up to the highest address block in the device. Block numbers may be described differently in the data sheet. For example, in a Flash device containing 64 blocks, the Flash device driver interface will always refer to the block with address offset 0 as block number 0, and to the last block as block number 63.

With the software driver interface, users can focus on writing the high-level code required for their particular application. The high-level code accesses the Flash memory by calling the low-level code. As a result, it is not necessary to understand the details of the special command sequences. The resulting source code is both simpler and easier to maintain.

Code developed using the provided drivers can be broken into three layers:

- Hardware-specific bus operations
- Low-level code
- High-level code written by the user

The low-level code requires hardware-specific READ and WRITE bus operations in C to communicate with the M29EW device. The implementation of these operations is hardware platform dependent as it depends on the microprocessor on which the C code runs and on the location of the memory in the microprocessor's address space.

C drivers that are suitable for the current hardware platform must be written by the user. The low-level code takes care of issuing the correct WRITE operation sequence for each command and of interpreting the information received from the device during programming and erasing.

The high-level code written by the user accesses the memory devices by calling the low level code. As a result, the code used is simple and easier to maintain. Also, the user's high-level code is easier to apply to other Micron Flash memory devices.

When developing an application, complete the following steps:

- Write a simple program to test the low-level code provided and verify that it operates as expected in the target hardware and software environments.
- Write the high-level code for the desired application. The application will access the Flash memory device by calling the low-level code.
- Thoroughly test the complete source code of the application.

Porting the Drivers to the Target System (User Change Area)

All changes to the software driver are made in the header file. A designated area called the user change area contains the items required to port the software driver to new hardware:

Basic Data Types

Check whether the compiler to be used supports the following basic data types, as described in the source code, and change it where necessary.

```
typedef unsigned char    NMX_uint8;    (8 bits)
typedef char             NMX_sint8;    (8 bits)
typedef unsigned short   NMX_uint16;   (16 bits)
typedef short            NMX_sint16;   (16 bits)
typedef unsigned int     NMX_uint32;   (32 bits)
typedef int              NMX_sint32;   (32 bits)
```

Device Type

Choose the correct device by using the appropriate define statement:

```
#define USE_M29EW128_8 and USE_M29EW128_16
#define USE_M29EW256_8 and USE_M29EW256_16
#define USE_M29EW512_8 and USE_M29EW512_16
```

Flash Memory Location

BASE_ADDR is the start address of the Flash memory. It must be set according to the target system to access the Flash memory at the correct address. This value is used by the FlashRead() and FlashWrite() functions. The default value is set to zero and must be adjusted appropriately:

```
#define BASE_ADDR ((volatile uCPUBusType*)0x00000000)
```

Flash Configuration

Choose the correct Flash memory configuration:

```
#define USE_8BIT_CPU_ACCESSING_1_8BIT_FLASH
```

This define statement supports a board configuration containing a CPU with an 8-bit data bus and a single 8-bit Flash memory device connected to it.

```
#define USE_16BIT_CPU_ACCESSING_2_8BIT_FLASH
```

This define statement supports a board configuration containing a CPU with a 16-bit data bus and two 8-bit Flash memory devices connected to it.

```
#define USE_32BIT_CPU_ACCESSING_4_8BIT_FLASH
```

This define statement supports a board configuration containing a CPU with a 32-bit data bus and four 8-bit Flash memory devices connected to it.

```
#define USE_16BIT_CPU_ACCESSING_1_16BIT_FLASH
```

This define statement supports a board configuration containing a CPU with an external 16-bit data bus with a single 16-bit Flash memory device connected to it.

```
#define USE_32BIT_CPU_ACCESSING_2_16BIT_FLASH
```

This define statement supports a board configuration containing a CPU with an external 32-bit data bus with two 16-bit Flash memory devices connected to it.

Timeout

Timeouts are implemented in the loops of the code to provide an exit for operations that would otherwise never terminate. There are two possibilities:

The ANSI library functions declared in time.h exist. If the current compiler supports time.h, the define statement TIME_H_EXISTS must be activated to prevent any change in timeout settings due to the performance of the current evaluation hardware.

```
#define TIME_H_EXISTS
```

The Option COUNT_FOR_A_SECOND: If the current compiler does not support time.h, the define statement TIME_H_EXISTS cannot be used. In this case, the COUNT_FOR_A_SECOND value must be defined so as to create a 1-second delay. For example, if 100,000 repetitions of a loop are needed to give a time delay of 1 second, then COUNT_FOR_A_SECOND should have the value 100,000.

```
#define COUNT_FOR_A_SECOND (chosen value)
```

Note: This delay depends on hardware performance and should be updated every time the hardware is changed.

This driver has been tested with a certain configuration and other target platforms may have other performance data. It may therefore be necessary to change the COUNT_FOR_A_SECOND value. It is up to the user to implement the correct value to prevent the code from timing out too early and allow correct completion.

Pause

The Flashpause() function is used in the code to generate delays necessary for the correct operation of the Flash device. There are two options:

The Option ANSI Library functions declared in time.h exist. If the current compiler supports time.h, the define statement TIME_H_EXISTS must be activated to prevent any change in timeout settings due to the performance of the current evaluation hardware.

```
#define TIME_H_EXISTS
```

The Option COUNT_FOR_A_MICROSECOND. If the current compiler does not support time.h, the define statement TIME_H_EXISTS cannot be used.

In this case, the COUNT_FOR_A_MICROSECOND value must be defined so as to create a 1-microsecond delay.

A value depending on a "While(count-- != 0);" loop must be found that generates the necessary delay.

An approximate value can be given by using the clock frequency of the test platform. This means that if an evaluation board operating at 200MHz is used, the value for COUNT_FOR_A_MICROSECOND will be 200.

The exact value can only be found using a logic state analyzer.

```
#define COUNT_FOR_A_MICROSECOND (chosen value)
```

Note: This delay depends on hardware performance and must be updated each time the hardware is changed.

This driver has been tested with a certain configuration and other target platforms may have other performance data. It may be necessary to change the value.

Additional Subroutines

```
#define VERBOSE
```

In the software driver, the VERBOSE define statement is used to activate the FlashErrStr() function, which generates a text string describing the return code from the Flash memory.

Additional Considerations

The access timing data for a Flash memory device can sometimes be problematic. It may be necessary to change the FlashRead() and FlashWrite() functions if they are not compatible with the timings of the target hardware. These problems can be solved with a logic state analyzer.

The programmer must take extra care when the device is accessed during an interrupt service routine. When the device is in Read mode, interrupts can freely read from the device. Interrupts that do not access the device may be used during all functions.

C Library Functions

The software library described in this technical note provides the user with source code for the following functions:

Table 1: Function Names and Descriptions

Function	Description
Flash()	Accesses all device functions. It acts as the main Flash memory interface. This function is available on all software drivers written in the Flash device driver format and should be used exclusively. Any functionality unsupported by the device can be detected and malfunctions can thus be avoided. **Note:** The other functions are listed to offer a second-level interface when enhanced performance is required. Within the Flash device driver the functions are always used in the same way. This means that the function interface (names, return codes, parameters, data types) remains unchanged, regardless of the Flash memory device.
Flash(BlockErase, ParameterType)	Erases one block.
Flash(CheckCompatibility, ParameterType)	Checks the compatibility of the Flash device.
Flash(Program, ParameterType)	Programs an array of elements.
Flash(Read, ParameterType)	Reads from the Flash device.
Flash(ReadCfi, ParameterType)	Reads CFI information from the Flash device.
Flash(ReadDeviceId, ParameterType)	Gets the Device ID from the device.
Flash(ReadManufacturerCode, ParameterType)	Gets the manufacturer code from the device.
Flash(Reset, ParameterType)	Resets the Flash for normal memory access.
Flash(Resume, ParameterType)	Resumes a suspended ERASE.
Flash(SingleProgram, ParameterType)	Programs a single element.
Flash(Suspend, ParameterType)	Suspends an ERASE.
Flash(Write, ParameterType)	Writes a value to the Flash device.
FlashBlockErase()	Erases a block in the device. A block cannot be erased when it is protected. Attempting to do so generates an error.
FlashCheckCompatibility()	Checks the Flash memory device for compatibility.
FlashChipErase()	Erases the entire device. Protected blocks will be not erased.
FlashEnterExtendedBlock()	Places the device in extended block mode, making it possible to read from and write to the Extended Block.
FlashBlockEraseError()	Generates a text string describing the detected error.
FlashExitExtendedBlock()	Causes the device to exit the extended block mode to prevent the extended block from being accessed.
FlashInit()	Used to initialize the driver. The FlashInit() function sets the internal variables and the Flash device description. It should be called before using any other function.
FlashMultipleBlockErase()	Erases selected blocks belonging to the same Flash memory bank.

Table 1: Function Names and Descriptions (Continued)

Function	Description
FlashProgram()	Programs arrays of elements to the Flash memory device. Only previously erased elements can be programmed reliably. Locked blocks cannot be programmed and PROGRAM operations cannot be performed when V_{PP} is invalid.
FlashReadCfi()	Checks whether the CFI is supported and then reads the CFI data at the specified offset.
FlashReadDeviceId()	Reads the first device code of the Flash memory device.
FlashReadMultipleDeviceId()	Reads the chosen device code of the Flash memory device.
FlashReadManufacturerCode()	Reads the manufacturer code of the Flash memory device.
FlashReset()	Resets the device to read mode. Note that there should be no need to call this function under normal operation as all other software library functions leave the device in this mode.
FlashSingleProgram()	Programs a single element. Only a previously erased element can be programmed reliably.
FlashUnlockBypass()	Enters the unlock bypass program mode. There is no need to call this function when VPP is applied to the VPP/Write Protect pin.
FlashUnlockBypassProgram()	Programs a single Flash memory location in unlock bypass mode.
FlashUnlockBypassReset()	Resets the device to read mode.
FlashUnlockBypassBlockErase()	Erases the block.
FlashUnlockBypassChipErase()	Erases the chip.
FlashUnlockBypassBufferProgram()	Programs a buffer.
FlashSuspend()	Suspends a PROGRAM/ERASE operation.
FlashResume()	Resumes a suspended PROGRAM/ERASE operation.
FlashWriteToBufferProgram()	Makes use of the device's 512-word/1024-byte write buffer to speed up programming.
FlashBufferProgramConfirm()	Confirms a WRITE TO BUFFER PROGRAM command and to program the N+1 words/bytes loaded in the write buffer by this command.
FlashBufferProgramAbort()	Must be issued to abort the WRITE TO BUFFER PROGRAM.
FlashCheckBlockProtection()	Returns the protection mode.
FlashReadExtendedBlockVerifyCode()	Verifies whether the extended memory block was locked or not locked by the manufacturer.
FlashCheckProtectionMode()	Reads the protection mode.
FlashSetNVProtectionMode()	Sets the device in volatile protection mode.
FlashSetPasswordProtectionMode()	Sets the device in password protection mode.
FlashSetExtendedBlockProtection()	Sets the extended ROM in protection mode.
FlashSetNVPBLockBit()	Sets the nonvolatile protection bit. Note that there is only one NVPB lock bit per device that is volatile. This bit can be cleared by software means in password protection mode or by hardware means in nonvolatile protection mode (similar to a power up or a hardware reset).
FlashCheckNVPBLockBit()	Checks the nonvolatile protection bit.

Table 1: Function Names and Descriptions (Continued)

Function	Description
FlashClearBlockNVPB()	Clears the nonvolatile modify protection bit of all blocks. All blocks' nonvolatile protection bit will be set and then cleared to prevent damage.
FlashSetBlockNVPB()	Sets the nonvolatile modify protection bit of a block.
FlashCheckBlockVPB()	Checks the volatile protection bit status of a block.
FlashClearBlockVPB()	Clears the volatile protection bit of a block.
FlashSetBlockVPB()	Sets the volatile protection bit status of a block.
FlashPasswordProgram()	Sets the password (64 bit) for password protection mode.
FlashVerifyPassword()	Verifies the password (4 word).
FlashPasswordProtectionUnlock()	Clears the NVPB lock bit under password protect mode.
FlashExitProtection()	Exits protection mode.

The functions provided in the software library rely on the user implementing the hardware-specific bus operations and on access timings to communicate properly with the Flash memory device. If changes to the software driver are necessary, only the two following functions must be changed:

Function	Description
FlashRead()	Reads a value from the Flash memory device.
FlashWrite()	Writes a value to the Flash memory device.

Getting Started (Example Quicktest)

To test the source code in the target system, start by reading from the M29EW device. If it is erased, only FFFFh data should be read. Then, read the manufacturer and device codes and verify that they are correct. If these functions work, the other functions are likely to work, too. However, all the functions should be tested thoroughly.

To start, write a function main() and include the C file as described in the following example. All Flash memory functions can be called and executed within the main() function.

The following example shows a check of the device identifiers (device code, manufacturer code) and a simple BLOCK ERASE command.

```
#include "M29EW_32Mb-2Gb_device_driver.h"

void main(void) {
    ParameterType fp;         /* Contains all Flash Parameters */
    ReturnType rRetVal;       /* Return Type Enum */
    uCPUBusType ucDeviceId;   /*To store the device Id */

    Flash(ReadManufacturerCode, &fp);
    printf("Manufacturer Code: %08Xh\r\n",
        fp.ReadManufacturerCode.ucManufacturerCode);

    Flash(ReadDeviceId, &fp);
    printf("Device Code: %08Xh\r\n",
        fp.ReadDeviceId.ucDeviceId);

    fp.BlockErase.ublBlockNr = 10;    /* block number 10 will be erased*/
    rRetVal = Flash(BlockErase, &fp); /* function execution */

    /* to read the second device Id */
    rRetval = FlashReadMultipleDeviceId(1, &ucDeviceId);
    printf("Device Code: %08Xh\r\n",ucDeviceId);
} /* EndFunction Main */
```

Software Limitations

The described software implements the full set of M29EW features. When an error occurs, the software simply returns the error message. It is up to the user to decide what to do. The user can either try the command again or, if necessary, replace the device.

Revision History

Rev. H – 04/15

- Revised to meet Micron styles and standards, including the addition of a revision history topic, previously not included
- Revised introduction to be consistent with Micron style
- Revised .c and .h file names to be consistent with code

Rev. A through G – 04/11

- Initial release
- Revisions B through G not captured in a revision history topic