Technical Note

Software Device Drivers for Micron® M29W256 NOR Flash Memory

Introduction

This technical note describes the library source code in C for the M29W256 parallel NOR Flash memory device using the Flash software device driver interface.

The source code is available from your Micron distributor. The M29W256.c and M29W256.h files contain libraries for accessing the M29W256 Flash memory device.

This technical note also includes an overview of the programming model for the M29W256 device. It describes the operation of the memory device and provide a basis for understanding and modifying the accompanying source code.

The source code is written to be as platform independent as possible. It requires minimal changes by the user in order to compile and run. The technical note explains how the user should modify the source code for individual target hardware. The source code contains comments throughout, explaining how it is used and why it has been written the way it has.

This technical note does not replace the M29W256 data sheet. It refers to it throughout, and it is necessary to have a copy of the data sheet to follow some of the explanations. The software has been tested on a target platform, and is usable in C and C++ environments. It is small in size and can be applied to any target hardware.

M29W256 Programming Model

The M29W256 256Mb nonvolatile Flash memory device 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 M29W256 has one extra 128-word block (extended memory block) that can only be accessed using a dedicated command. The Extended Block can be protected, which is useful for storing security information. However, the protection is not reversible. This 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 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 Command Interface of the memory. An on-chip PROGRAM/ERASE Controller (P/E.C.) takes care of the 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 M29W256 features are available via the 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 that modify the data stored or the device's behavior.

Only certain special WRITE operation sequences are recognized as commands by the M29W256. The various commands recognized by the devices are listed in the Commands Tables provided in the data sheet. The main commands can be classified as:

- READ/RESET
- AUTO SELECT
- BLOCK ERASE
- CHIP ERASE
- PROGRAM
- FAST PROGRAM
  - BUFFER PROGRAM
  - UNLOCK BYPASS
  - ENHANCED BUFFER PROGRAM
- PROGRAM/ERASE SUSPEND AND RESUME
- READ COMMON FLASH INTERFACE QUERY
- CHIP UNPROTECT
- VOLATILE BLOCK PROTECT
- NON VOLATILE BLOCK PROTECT
- PASSWORD BLOCK PROTECT
- ENTER/EXIT EXTENDED BLOCK

The READ/RESET command returns the M29W256 to Read Mode where it behaves as a ROM. In this mode, a READ operation will output 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 Auto Select mode, the system can read the manufacturer code, device code, 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. It may therefore 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 some block are protected, then these are 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 100s, 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 M29W256 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 the user to access information stored in the M29W256 immediately 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 M29W256 offers a set of Fast Program commands to improve the programming throughput:

- WRITE TO BUFFER PROGRAM makes use of the device's 32-word/64-byte write buffer to speed up programming.
- WRITE TO ENHANCED BUFFER PROGRAM makes use of the device's 256-word write buffer to speed up programming. Note that the ENHANCED BUFFER PROGRAM command set is available only in x16 mode.
- UNLOCK BYPASS is used to place the device in Unlock Bypass mode. When the device enters the Unlock Bypass mode, the two initial unlock cycles required in the standard program command sequence are no longer needed, and only two WRITE cycles are required to program data, instead of the normal four cycles. This results in a faster total programming time.

The READ CFI QUERY command of the device is used to read data from the Common Flash Interface (CFI), allowing the user to identify the number of blocks in the Flash memory and the block addresses. The interface also contains information relating to the typical and maximum PROGRAM and ERASE times. This allows the user to implement software timeouts and prevents waiting indefinitely for a defective Flash memory to finish programming or erasing. For further information about the CFI, please refer to the CFI specification available from the internet site (http://www.jedec.org) or from your Micron distributor.

The devices are shipped with all blocks unprotected. The block protection status can be read, for example, by performing a read electronic signature. The M29W256 has three different software protection modes: volatile protection, nonvolatile protection, and password protection.

On first use, all parts default to operate in nonvolatile protection mode and the you are free to activate the nonvolatile or the password protection mode. The desired protection mode is activated by setting a bit of the Lock Register. For details, refer to the data sheet.

Status Register

The M29W256 discrete device has one Status Register. The Status Register provides information on the current or previous PROGRAM or ERASE operations executed. 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. It is also read during and ERASE SUSPEND when an address within a block being erased is accessed.

Programming Command Sequence Example

The Commands tables provided in the M29W256 data sheet describe the WRITE operation sequences recognized as valid commands by the PROGRAM/ERASE Controller.

As an example, consider the programming of 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 NMX _uint16 is defined as the following 16-bit value:

```
typedef unsigned short NMX_uint16
```

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 M29W256 is mapped to the address 0000h in the microprocessor address space. In practice, the Flash memory is likely to have a base offset that will have to 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 or not the operation has completed.

Using the Software Driver

General Considerations

The software driver described in this technical note are intended to simplify the process of developing an application code in C for the Micron M29W256 Flash memory device. This software driver supports the Flash device driver interface that is implemented in all new software drivers. As a result, future device changes will not necessarily lead to code changes in application environments.

Note that to meet compatibility requirements, the standard software driver interface allocates numbers to each block in a Flash memory 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 sheets. 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.

This technical note gives a summary description of the Flash device driver interface. A complete description is available from your Micron distributor.

With the software driver interface, you can focus on writing the high-level code required for their particular applications. The high-level code accesses the Flash memory by calling the low-level code, so you do not have to concern yourself with 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 down 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 M29W256 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 must be written that are suitable for the current hardware platform. The low-level code takes care of issuing the correct WRITE operation sequence for each command, and of interpreting the information received from the devices during programming and erasing.

The high-level code you write accesses the memory devices by calling the low-level code. In this way, the code used is simple and easy to maintain. Another consequence is that the your high-level code is easier to apply to other Micron Flash memory devices.

When developing an application:
1. Write a simple program to test the low-level code provided, and verify that it operates as expected in the user's target hardware and software environments.
2. Write the high-level code for the desired application. The application will access the Flash memory device by calling the low-level code.
3. Thoroughly test the complete source code of the application.

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

All typical changes to the software driver can be found 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: Checks whether the compiler to be used supports the following basic data types, as described in the source code, and changes 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: Chooses the correct device by using the appropriate define statement:

```
#define USE_M29W256_8 and USE_M29W256_16

#define USE_M29W256_8 and USE_M29W256_16

#define USE_M29W256_8 and USE_M29W256_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 0, and must be adjusted appropriately:

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

Flash Configuration: Chooses 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 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 should be activated. This will 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 the hardware performance and should therefore 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. You must 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 should be activated. This will 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 has to 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. In other words, 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 that this delay depends on the hardware performance and must therefore 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. Therefore, the value may have to be changed.

Additional Subroutines: #define VERBOSE

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

Additional Considerations: The access timing data for Flash memory devices 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 source code for the following functions:

- **Flash**() is used to access all device functions, and 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 Flash memory 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.

- **Flash(BlockErase, ParameterType)** is used to erase one block.
- **Flash(CheckCompatibility, ParameterType)** is used to check the compatibility of the Flash.
- **Flash(Program, ParameterType)** is used to program an array of elements.
- **Flash(Read, ParameterType)** is used to read from the Flash device.
- **Flash(ReadCfi, ParameterType)** is used to read CFI information from the Flash device.
- **Flash(ReadDeviceId, ParameterType)** is used to get the device ID from the device.
- **Flash(ReadManufacturerCode, ParameterType)** is used to get the manufacture code from the device.
- **Flash(Reset, ParameterType)** is used to reset the Flash for normal memory access.
- **Flash(Resume, ParameterType)** is used to resume a suspended erase.
- **Flash(SingleProgram, ParameterType)** is used to program a single element.
- **Flash(Suspend, ParameterType)** is used to suspend an erase.
- **Flash(Write, ParameterType)** is used to write a value to the Flash device.
- **FlashBlockErase**() is used to erase a block in the device. A block cannot be erased when it is protected. Attempting to do so generates an error.
- **FlashCheckCompatibility**() is used to check the Flash memory for compatibility.
- **FlashChipErase**() is used to erase 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**() is used to generate a text string describing the detected error.
- **FlashExitExtendedBlock**() causes the device to exit the Extended Block mode so that the Extended Block can no longer be accessed.
- **FlashMultipleBlockErase**() is used to erase selected blocks belonging to the same Flash memory bank.

- **FlashProgram**() is used to program arrays of elements to the Flash memory. 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**() is used to check whether the Common Flash Interface is supported, and then to read the CFI data at the specified offset.
- **FlashReadDeviceId**() is used to read the first device code of the Flash device.
- **FlashReadMultipleDeviceId**() is used to read the chosen device code of the Flash device.
- **FlashReadManufacturerCode**() is used to read the Manufacturer Code of the Flash device.
- **FlashReset**() is used to reset 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**() is used to program a single element. Only a previously erased element can be programmed reliably.
- **FlashUnlockBypass**() is used to enter the unlock Bypass Program mode. There is no need to call this function when V_{PP} is applied to the V_{PP}/Write Protect pin.
- **FlashUnlockBypassProgram**() is used to program a single Flash memory location in Unlock Bypass mode.
- **FlashUnlockBypassReset**() is used to reset the device to Read mode.
- **FlashUnlockBypassBlockErase**() is used to erase the block.
- **FlashUnlockBypassChipErase**() is used to erase the chip.
- **FlashUnlockBypassBufferProgram**() is used to program a buffer.
- **FlashSuspend**() is used to suspend a PROGRAM/ERASE operation.
- **FlashResume**() is used to resume a suspended PROGRAM/ERASE operation.
- **FlashWriteBufferProgram**() makes use of the device's 32-word/64-byte write buffer to speed up programming.
- **FlashBufferProgramConfirm**() is used to confirm 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.
- **FlashWriteEnhancedBufferProgramEnter**() makes use of the device's 256-word write buffer to speed up programming.
- **FlashEnhancedBufferProgramConfirm**() is used to confirm a WRITE TO ENHANCED BUFFER PROGRAM command and to program the N+1 words/bytes loaded in the write buffer by this command.
- **FlashEnhancedBufferProgramExit**() must be issued to abort the WRITE TO ENHANCED BUFFER PROGRAM.
- **FlashCheckBlockProtection**() is used to return the protection mode.
- **FlashReadExtendedBlockVerifyCode**() is used to verify if the Extended Memory Block was locked or not locked by the manufacturer.
- **FlashCheckProtectionMode**() is used to read the Protection Mode.
- **FlashSetNVProtectionMode**() is used to set the device into Volatile Protection Mode.
- **FlashSetPasswordProtectionMode**() is used set the device into Password Protection mode.
- **FlashSetExtendedBlockProtection**() is used to set the extended ROM in protection mode.

- **FlashSetNVPBLockBit**() is used to set 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**() is used to check the Nonvolatile Protection bit.
- **FlashClearBlockNVPB**() is used to clear the Nonvolatile Modify Protection Bit of all blocks. All blocks' Nonvolatile Protection Bit will be set and then cleared to prevent damage.
- **FlashSetBlockNVPB**() is used to set the Nonvolatile Modify Protection bit of a block.
- **FlashCheckBlockVPB**() is used to check the Volatile Protection Bit status of a block.
- **FlashClearBlockVPB**() is used to clear the Volatile Protection bit of a block.
- **FlashSetBlockVPB**() is used to set the Volatile Protection Bit status of a block.
- **FlashPasswordProgram**() is to set the password(64 bit) for password protection mode.
- **FlashVerifyPassword**() is used to verify the password(4 word).
- **FlashPasswordProtectionUnlock**() is used to clear the NVPB Lock Bit under Password Protect mode.
- **FlashExitProtection**() is used to exit Protection mode.

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

- **FlashRead**() is used to read a value from the Flash device.
- **FlashWrite**() is used to write a value to the Flash device.

Getting Started (Example Quicktest)

To test the source code in the target system, start by reading from the M29W256 devices. If it is erased, only FFFFh data should be read. Then, read the manufacturer and device codes and check 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.

```c
#include "M29W256.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 M29W256 features. When an error occurs, the software simply returns the error message. It is up to you to decide what to do. You can either try the command again or replace the device if necessary.

Conclusion

The Flash device driver interface allows changeable Flash configurations, compiler-independent data types, and a unique access mode for a broad range of Flash devices.

Moreover, applications supporting the Flash device driver standard can implement any Flash device with the same interface without any code change. Recompiling with a new software driver is all that is needed to control a new device.