Technical Note

Software Device Drivers for Micron® M29Fxx NOR Flash Memory

Introduction

This technical note provides library source code in C for M29Fxx parallel NOR Flash memory using the Flash software device driver interface.

The M29F is available in 8-bit or 16-bit bus widths and as a top and bottom boot block device. This technical note supports the devices in all possible variations, are referred to as the "M29F" within this document, except where stated otherwise.

The C code drivers work on a layer above the hardware, and can be used successfully over either bus interface.

Also included in this application note is an overview of the programming model for the M29F. This will familiarize the reader with the operation of the memory devices and provide a basis for understanding and modifying the accompanying source code.

The source code is written to be as platform independent as possible and requires minimal changes to compile and run. This document explains how to modify the source code for the individual target hardware. All related source code contains comments explaining how it is used, and why it has been written the way it has.

This technical note does not replace the M29Fxx data sheet. It refers to the data sheet throughout, and it is necessary to have a copy to follow some of the explanations. The software and accompanying documentation 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.

M29F Programming Model

The M29F is a 16Mb (2Mb x8 or 1Mb x16) nonvolatile NOR Flash memory that can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is down into 35 blocks, each 64KB in size. The first or the last 64KB have been divided into four additional blocks: a boot block (16KB), two parameter blocks (8KB), and a block with a bigger quantity (32KB), called the main block. Each block of the whole device can be erased individually.

Included in the device is a PROGRAM/ERASE Controller. With first generation Flash memory devices the software had to program manually all of the words to 0000h, before erasing to FFFFh, using special programming sequences. The PROGRAM/ERASE Controller in the M29F device allows a simpler programming model to be used by taking care of all necessary steps required to ERASE and PROGRAM the memory. This has led to improved reliability so that in excess of 100,000 PROGRAM/ERASE cycles are guaranteed per block on the device.

Note: Data with the current CPU data bus width is referred to as elements throughout the document unless otherwise specified. Due to the flexibility of the software driver, the size of an element depends on the current configuration.

Bus Operations and Commands

Most of the functionality of the M29F is available via the two standard bus operations: READ and READ. READ operations retrieve data or status information from the device. WRITE operations are interpreted by the device as commands, which modify the behavior of the device or the data stored.

Only certain special sequences of WRITE operations are recognized as commands by the M29F. The various commands recognized by the M29F are listed in the Instructions table in the data sheet. The main commands can be grouped as follows:

1. READ
2. READ ELECTRONIC SIGNATURE
3. ERASE
4. PROGRAM (WRITE and PROGRAM)

The READ command returns the M29F to Read mode, where it behaves as a ROM. In this state, a READ operation outputs onto the data bus the data stored at the specified address of the device. The READ ELECTRONIC SIGNATURE command places the device in a mode that allows the user to read the Electronic Signature of the device. The Electronic Signature (manufacturer and device codes) is accessed by reading different addresses while in the Auto Select mode.

The ERASE command is used to set all the bits to 1 in every memory location in the selected block. All data previously stored in the erased block are lost. The ERASE command takes longer to execute than the other commands because an entire block is erased at once. Attempts to ERASE or PROGRAM a block without connected V_{PP} voltage do not modify the contents of the memory.

The WORD PROGRAM and the MULTIPLE WORD PROGRAM command is used to modify the data stored at the specified addresses of the device. Programming modifies one or more words at a time.

Status Register

While the M29F is programming or erasing, a read from the device will output the Status Register of the PROGRAM/ERASE Controller. The Status Register provides valuable information about the most recent PROGRAM or ERASE command. The Status Register bits are described in the Status Register Bits Table of the M29F data sheet. Their main use is to give additional information about the Flash device when a PROGRAM or ERASE fails. After a successful PROGRAM or ERASE operation, the device returns automatically into Read Mode.

Completion of the PROGRAM or ERASE operation can be determined either from the polling bit (Status Register bit DQ7) or from the toggle bit (Status Register bit DQ6) by following the Data Polling Flow Chart or the Data Toggle Flow Chart in the data sheet. The library routines described in this application note use the Data Toggle Flow Chart. However, a function based on the Data Polling Flow Chart is also provided as an illustration. Programming or erasing errors are indicated by the error bit (Status Register bit DQ5) that is set to 1 when an error occurs. If a failure occurs, the command will not complete and READ operations will continue to output the Status Register bits until a READ/RESET command is issued to the device.

Detailed Example

The Command Table of the M29Fxx data sheet describes the sequences of WRITE operations that are recognized by the PROGRAM/ERASE Controller as valid commands.

As an example, consider the programming of the value 9465h to the address 03E2h for the M29F. The C programming language requires the user to write the following sequence (in C):

where NMX_uint16 is defined as the following 16-bit value:

```
typedef unsigned short NMX_uint16.
```

This example assumes that address 0000h of the M29F is mapped to address 0000h in the microprocessor address space. In practice, it is likely that the Flash will have a base offset that must be added to the address. While the device is programming the specified address, READ operations will access the Status Register bits. Status Register bit DQ5 will be 1 if an error has occurred. Bit DQ6 will toggle while programming is ongoing. Bit DQ7 will be the complement of the data being programmed

Using the Software Driver

General Considerations

The low-level functions (drivers) described in this technical note simplify the process of developing application code in C for the Micron M29F Flash memory device.

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

Note: 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 number 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 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 drivers
- High-level functions written by the user

The implementation, in C, of the hardware-specific READ and WRITE bus operations is required by the low-level drivers in order to communicate with the M29F device. This implementation is hardware platform dependent. It is dependent on which microprocessor the C code runs, and where in the microprocessor's address space the memory device is located.

You must write the C functions that are appropriate to the current hardware platform. The low-level drivers take care of issuing the correct sequences of WRITE operations for each command and of interpreting the information received from the device during programming and erasing. These drivers encode all the specific details of how to issue commands and how to interpret the Status Register bits.

The high-level functions written by you access the memory device by calling the low-level functions.

When developing an application:

1. Write a simple program to test the low-level drivers provided and verify that these operate as expected on the target hardware and software environments.
2. Write the high-level code for the application, which accesses the Flash memory by calling the low-level drivers.
3. Test the complete application source code thoroughly.

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

All sensible changes to the software driver that must be considered can be found in the header file. There is one designated area called User Change Area, which contains the following items to port the software driver to a new hardware:

Basic Data Types: Check if 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_ M29FT_8
```

This define statement supports the M29F top boot block device in an 8-bit configuration.

or

```
#define USE_ M29FT_16
```

This define statement supports the M29F top boot block device in a 16-bit configuration.

or

```
#define USE_ M29FB_8
```

This define statement supports the M29WF bottom boot block device in an 8-bit configuration.

or

```
#define USE_ M29FB_16
```

This define statement supports the M29F bottom boot block device in a 16-bit configuration.

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 functions FlashRead() and FlashWrite(). The default value is set to 0, 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.

or

```
#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 single 8-bit Flash memory devices connected to it.

or

```
#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 16-bit Flash memory devices connected to it.

or

```
#define USE_16BIT_CPU_ACCESSING_1_16BIT_FLASH
```

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

or

```
#define USE_32BIT_CPU_ACCESSING_2_16BIT_FLASH
```

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

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

1. The Option ANSI Library functions declared in time.h exists.
   If the current compiler supports time.h, the define statement TIME_H_EXISTS should be activated. This ensures that the performance of the current evaluation hardware does not change the timeout settings.

   ```
   #define TIME_H_EXISTS
   ```

2. The Option COUNT_FOR_A_SECOND)
   If the current compiler does not support time.h, the define statement TIME_H_EXISTS cannot be used. To overcome this constraint, the value COUNT_FOR_A_SECOND must be defined in order 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, 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. It is up to you to implement the correct value to prevent the code from timing out too early and allow correct completion.

Pause Constant: The function Flashpause() is used in several areas of the code to generate a delay required for correct operation of the Flash device. There are two options provided:

1. 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 ensures that the performance of the current evaluation hardware does not change the timeout settings.

```
#define TIME_H_EXISTS
```

2. The Option COUNT_FOR_A_MICROSECOND

If the current compiler does not support time.h, the define statement TIME_H_EXISTS cannot be used. To overcome this constraint, the value COUNT_FOR_A_MICROSECOND must be defined to create a 1-microsecond delay.

Depending on a While(count-- != 0); loop, a value must be found that creates the necessary delay.

An approximate approach can be given by using the clock frequency of the test platform. That means if an evaluation board with 200 Mhz is used, the value for COUNT_FOR_A_MICROSECOND would be 200.

The real 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 should 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 this software driver, the define statement VERBOSE 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 the 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 provided with this technical note provides the source code for the following functions:

Flash() is used to access all chip 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.

This function is guaranteed. Any unsupported functionality of the Flash memory can be detected and, therefore, malfunction can be avoided.

Note: The other functions are listed to offer a second-level interface for enhanced performance. These functions are guaranteed only in other software drivers using the same functionality. Within the Flash device driver, the functions are used always in the same way. This means that the function interface (names, return codes, parameters, data types) remains the same, independent of the Flash memory device.

FlashBlockErase() is used to erase a block in the Flash. If the selected block is protected, this is ignored and no error condition is given. During the BLOCK ERASE operation the memory will ignore all commands except the ERASE SUSPEND and READ/RESET commands.

FlashBlockProtect() is used to protect one block in the Flash. This function uses the In System Technique to protect one block. This technique requires a high voltage level on Reset/Blocks Temporary Unprotect pin RP. Otherwise the function returns an error condition.

FlashCheckBlockProtection() is used to check the block protection status.

FlashCheckCompatibility() is used to check if the device Flash is compatible with the current device and software driver.

FlashChipErase() is used to erase the whole chip. Only unprotected blocks can be deleted. Protected single blocks will be ignored and left unchanged. If all blocks are protected, the CHIP ERASE command will leave all blocks unchanged.

FlashChipUnprotect() is used to unprotect the whole chip. This function uses the In System Technique. This technique requires a high voltage level on Reset/Blocks Temporary Unprotect pin RP. Otherwise the function returns an error condition.

FlashMultipleBlockErase() can be used to erase a list of one or more blocks. Only unprotected blocks can be deleted. Protected single blocks will be ignored and left unchanged. If all blocks of the list are protected, the BLOCK ERASE operation appears to start, but will leave the data within all the blocks in the list unchanged. During the BLOCK ERASE operation, the memory will ignore all commands except the ERASE SUSPEND and READ/RESET commands.

FlashProgram() is used to program data arrays into the Flash. Note that the PROGRAM command cannot change a bit set at 0 back to 1. This case will return an error code. An ERASE command must be used to set all bits in a block or in the whole memory from 0 to 1. This function internally uses the commands UNLOCK BYPASS, UNLOCK BYPASS PROGRAM, and UNLOCK BYPASS RESET.

FlashResume() is used to issue the ERASE RESUME command.

FlashSuspend() is used to issue the ERASE SUSPEND command.

FlashRead() is used to read a value from the Flash device.

FlashReadCfi() is used to read data from the CFI.

FlashReadDeviceId() is used to read the device code. This function also ensures that all device configurations (eventually inserted on the board) reply equally.

FlashReadManufacturerCode() is used to read the manufacturer code.

FlashReset() is used to reset the device into the 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() can be used to program one element to the memory array. Note that the PROGRAM command cannot change a bit set at 0 back to 1. This case will return an error code. An ERASE Command must be used to set all the bits in a block or in the whole memory from 0 to 1.

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

FlashRead() is used to read a value (element) from the Flash memory.

FlashWrite() is used to write a value (element) to the Flash memory.

Examples of all these functions are provided in the source code (c1649.c).

Getting Started (Example Quicktest)

To test the source code in the target system, start by reading from the M29F device. If it is erased, then only FFh data (or FFFFh for 16-bit data, single configuration) should be read. Next, read the manufacturer code and device ID and check that they are correct.

If these functions work, it is highly likely that all the other functions work, too. However, they should all be tested thoroughly anyway.

To start, write a function main() and to include the C file as described in the following code example. Within the main function, all the Flash memory functions can be called and executed. The following example shows a check of the device identifiers (device ID, manufacturer code) and a simple BLOCK ERASE command.

```
#include "M29Fxx.h"


void main(void) {
    ParameterType fp;   /* Contains all Flash Parameters */
    ReturnType rRetVal; /* Return Type Enum */

    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 */

} /* EndFunction Main */
```

Software Limitations

The software provided does not implement a full set of the M29Fxx's functionality. When an error occurs, the software simply returns an error message. It is left to the user to decide what to do. Either the command can be tried again, or if necessary, the device may need to be replaced.

Conclusion

The M29F 5V Flash memory is an ideal product for automotive and other computer systems. It is able to be easily interfaced to microprocessors and driven with simple software drivers written in the C language.

With the Flash device driver interface, changeable Flash memory configurations, compiler-independent data types, and a unified access for a broad range of Flash memory devices can be introduced.

Moreover, applications supporting the Flash device driver standard can use all Flash memory devices with the same interface without any code change. Simply recompiling with a new software driver is all that is necessary to control the new device.

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