Technical Note

N25Q Serial NOR Flash Memory Software Device Drivers

Introduction

This technical note provides a description of the library source code in C for the N25Q Serial NOR Flash devices, which use interfaces similar to the Flash software device driver interface specification. The N25Q.c and N25Q.h files contain libraries for accessing the supported Serial NOR Flash devices. Micron's Serial NOR Flash devices, including the supported devices, are available in different configurations and densities. Valid part numbers are at Micron’s part catalog (www.micron.com).

This technical note does not duplicate or replace information from the N25Q data sheets. It refers to the data sheets throughout. It is necessary to have a copy of the appropriate data sheet to understand explanations.

This technical note provides information for modifying the accompanying source code. The source code is written to be as platform-independent as possible, and requires some changes by the user to compile and run.

The technical note also explains how the source code should be modified for individual target hardware. The source code contains comments throughout, which explain how it is used and why it has been written the way it has.

The software supplied with this documentation has been tested on a target platform and can be used in C and C++ environments. It is small in size and can be applied to any target hardware.

Using the Software Driver

The software driver simplifies the process of developing application code in C. This software driver is based on 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.

With the software driver interface, users can focus on writing the high-level code required for their particular applications. The high-level code accesses the device by calling the low-level code. Users do not have to concern themselves with the details of the special instruction 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 register READ and WRITE operations in C to communicate with the N25Q device. The implementation of these operations is hardware-platform dependent because it depends on the microprocessor and microcontroller on which the C code runs and on the location of the memory in the microprocessor's address space. The user must write the C code that is suitable for the current hardware platform. However, a guiding framework is provided in the Serialize.h and Serialize.c files. Those files are located in the Serialize Function for the SPI Controller file, which can be downloaded at:

micron.com/products/support/nor-flash-software

The high-level code written by the user accesses the devices by calling the low-level code. In this way, the code used is simple and easy to maintain. Another consequence is that the user's high-level code is easier to apply to other Serial NOR Flash 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 target hardware and software environments.
2. Write the high-level code for the desired application. The application will access the Serial NOR Flash device by calling the low-level code.
3. Thoroughly test the complete source code of the application.

Porting the Driver (User Change Area)

All of the changes to the software driver can be found in the header file. The designated area (called "the user change area") contains the items described in the following sections, which are 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.

Table 1: Basic Data Typedefs

```
typedef unsigned char NMX_uint8; (8 bits)
typedef char NMX_sint8; (8 bits)
typedef unsigned short NMX_uint16; (16 bits)
typedef unsigned short NMX_uint16; (16 bits)
typedef shortNMX_sint16; (16 bits)
typedef unsigned int NMX_uint32; (32 bits)
typedef int NMX_sint32; (32 bits)
typedef int NMX_sint32; (32 bits)
```

Device Type

The correct device is automatically detected at run time based on READ ID commands. This detection occurs during the Driver_Init() function. No defines are necessary to use either a 8Mb, 16Mb, 32Mb, 64Mb, 128Mb, 128Mb (N25Q128A8X), 256Mb, or 512Mb device. In N25QxxA8x devices, both standard command set and new command set are available for use. To use the new command set, the following define must be added into N25Q.h:

```
#define SUPPORT_N25Q_STEP_B
```

Timeout

Timeouts are implemented in code loops to provide an exit to operations that otherwise would 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 prevents 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 therefore should be updated each time the hardware is changed.

This driver has been tested with a specific configuration and other target platforms may have other performance data. It may 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 enable correct completion. In accordance to the corresponding data sheet, a suitable timeout value is configured in each function where required.

Additional Subroutines

```
#define VERBOSE
```

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

C Library Functions

The table below provides the user with source code for the following functions.

Table 2: Function Names and Descriptions

Function	Description
Driver_Init()	Initializes the driver and automatically detects the device. This function should be called before all other functions. If the function returns the Flash_WrongType value, the device has not been recognized. (See Sample Code.)
BulkErase()	Erases the entire memory by sending a BULK ERASE command.
DieErase()	Erases a single die in the stacked devices by sending a DIE ERASE command.
DeepPowerDown()	Sets the device in the lowest power consumption mode (the deep power-down mode) by sending a DEEP POWER-DOWN command. After calling this routine, the device will not respond to any command except the RELEASE FROM DEEP POWER-DOWN command.
DualInputFastProgram()	Programs 256 bytes of data or less to the memory on two pins (pin DQ0 and pin DQ1) instead of only one by sending a DUAL INPUT FAST PROGRAM command.
DualOutputFastRead()	Reads data from the memory on two pins (pin DQ0 and pin DQ1) instead of only one by sending a DUAL OUTPUT FAST READ command.
Enter4ByteAddressMode()	The ENTER 4 BYTE ADDRESS MODE command enables the 4-byte address mode (256Mb N25Q devices only).
Exit4ByteAddressMode()	The EXIT 4 BYTE ADDRESS MODE command disables the 4-byte address mode (256Mb N25Q devices only).
FastRead()	Reads the device by sending a READ DATA BYTES AT HIGHER SPEED (FAST_READ) command. By design, the entire space can be read with one FAST_READ command by incrementing the start address and rolling to 0h automatically. This means that this function is across pages and sectors.
FlashReadDeviceIdentification()	Reads the manufacturer identification (20h) and device identification by sending a READ IDENTIFICATION command. Note that the last two command can be unified into a single function due to the fact that only a single command returns the device and manufacturer identification.
PageProgram()	Programs 256 bytes or less of data to the memory by sending a PAGE PROGRAM command. By design, this operation is effective within one page, that is XX00h to XXFFh. When XXFFh is reached, the address rolls over to XX00h automatically. This function assumes that the memory to be programmed has been previously erased, or that bits are only changed from 1 to 0.
Program()	Programs a chunk of data into the memory at one time. After successfully verifying the start address and checking the available space, this function programs data from the buffer to the memory sequentially by invoking FlashPageProgram(). This function automatically handles page boundary crosses, if any. Similar to FlashPageProgram(), this function assumes that the memory to be programmed has been previously erased, or that bits are only changed from 1 to 0.
ProgramEraseResume()	Resumes the PROGRAM/ERASE operation that was suspended by sending a SPI_FLASH_INS_PER command.
ProgramEraseSuspend()	Resumes the PROGRAM/ERASE operation that was suspended by sending a SPI_FLASH_INS_PES command.
ProgramOTP()	Programs the 64-byte OTP area by sending a PROGRAM OTP command.

Table 2: Function Names and Descriptions (Continued)

Function	Description
QuadInputExtendedFastProgram()	Programs 256 bytes or less of data and address to the memory on four pins by sending a QUAD INPUT FAST PROGRAM command. Similar to FlashPageProgram(), this function assumes that the memory to be programmed has been previously erased, or that bits are only changed from 1 to 0.
QuadInputFastProgram()	Programs 256 bytes or less of data to the memory on four pins by sending a QUAD INPUT FAST PROGRAM command. Similar to FlashPageProgram(), this function assumes that the memory to be programmed has been previously erased, or that bits are only changed from 1 to 0.
QuadInputOutputFastRead()	Reads data from the memory and send addresses on four pins instead of only one by sending a QUAD OUTPUT FAST READ command.
QuadOutputFastRead()	Reads data from the memory on four pins instead of only one by sending a QUAD OUTPUT FAST READ command.
Read()	Reads the device by sending a READ DATA BYTES (READ) command. By design, the entire space can be read with one READ command by incrementing the start address and rolling to 0h automatically. This means that this function is across pages and sectors.
ReadLockRegister()	Reads the status register by sending a READ LOCK REGISTER command.
ReadOTP()	Reads data from the OTP area by sending a READ OTP command.
ReadStatusRegister()	Reads the status register by sending a READ STATUS REGISTER command.
ReadVolatileConfigurationRegister()	The READ VOLATILE CONFIGURATION REGISTER command enables the volatile configuration register to be read.
ReleaseFromDeepPowerDown()	Takes the device out of deep power-down mode by sending a RELEASE FROM DEEP POWER-DOWN command.
SectorErase()	Erases an entire sector by sending a SECTOR ERASE command.
SubSectorErase()	Erases a sub-sector by sending a SUB-SECTOR ERASE command.
WriteDisable()	Resets the WEL bit by sending a WRITE DISABLE command.
WriteEnable()	Sets the WEL bit by sending a WRITE ENABLE command.
WriteLockRegister()	Writes the lock register by sending a WRITE LOCK REGISTER command.
WriteStatusRegister()	Writes the status register by sending a WRITE STATUS REGISTER command.
WriteVolatileConfigurationRegister()	The WRITE VOLATILE CONFIGURATION REGISTER command enables new values to be written to the volatile configuration register. Before it can be accepted, a WRITE VOLATILE CONFIGURATION REGISTER command must previously have been executed.

Sample Code

The following source code is a sample that shows how the driver is used. The driver performs READ, PROGRAM, and ERASE operations.

Table 3: Quick Test Sample Code

```
#include <stdio.h>
#include "N25Q.h"
#include "Serialize.h"

int main(int argc, char ** argv)
{
   FLASH_DEVICE_OBJECT fdo;            /* flash device object */
   ParameterType para;                 /* parameters used for all operation */
   ReturnType ret;                     /* return variable */
   NMX_uint8 rbuffer[16];              /* read buffer */
   NMX_uint8 wbuffer[16] =             /* write buffer */
        {
         0xBE, 0xEF, 0xFE, 0xED, 0xBE, 0xEF, 0xFE, 0xED,
         0xBE, 0xEF, 0xFE, 0xED, 0xBE, 0xEF, 0xFE, 0xED
        };

   SpiDriverInit();                    /* initialize your SPI interface */
   ret = Driver_Init(&fdo);            /* initialize the flash driver */
   if (Flash_WrongType == ret)
        {
         printf("Sorry, no device detected.\n");
         return -1;
        }

   fdo.GenOp.SectorErase(0);           /* erase first sector */

   para.PageProgram.udAddr = 0;        /* program 16 byte at address 0 */
   para.PageProgram.pArray = wbuffer;
   para.PageProgram.udNrOfElementsInArray = 16;
   fdo.GenOp.DataProgram(PageProgram, &para);

   para.Read.udAddr = 0;               /* read 16 byte at address 0 */
   para.Read.pArray = rbuffer;
   para.Read.udNrOfElementsToRead = 16;
   fdo.GenOp.DataRead(Read, &para);

   printf("The first device byte is: 0x%x\n", rbuffer[0]);  /* now rbuffer contains
                                                              written elements */
   return 0;
}
```

Software Limitations

The software described in this document does not implement all functionality of the N25Q device. When an error occurs, the software simply returns the error message. When this happens, the user can either try the command again or replace the device if necessary.

Conclusion

N25Q Serial NOR Flash devices are ideal products for embedded and other computer systems. They can be easily interfaced to microprocessors and driven with simple software drivers written in the C language.

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.

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

Revision History

Rev. H – 05/13

- Updated the description in the Using the Software Driver section; added a link to the location of the Serialize.h and Serialize.c files on micron.com

Rev. G – 12/12

- Added additional data sheet densities
- Removed outdated data sheet information and included Micron Web information to ensure access to accurate, updated data sheets

Rev. F – 02/12

- Added 512Mb device to list of supported devices
- Removed outdated data sheet information and included references to data sheets to ensure continuous accurate and updated information

Rev. E – 10/11

- Updated the file names in the introduction
- Removed the memory layout description from the Programming Model section
- Added ENTER 4-BYTE ADDRESS MODE and EXIT 4-BYTE ADDRESS MODE to the commands table
- Added Segment selection (NVCR<1>) and Address byte (NVCR<0>) to the nonvolatile configuration register bits table
- Added the Extended Address Register section
- Replaced the Flag Status Register Bits table with the table from the data sheet
- Removed the obsolete sample code section
- Updated the Device Type description in the porting section
- Removed the Flash() and ReadDeviceIdentification() functions
- Modified the C library function names to remove the "flash" prefix
- Added the Driver_Init(), Enter4ByteAddressMode(), and Exit4ByteAddressMode() C library functions
- Replaced the FlashReadManufacturerIdentification() function with FlashReadDeviceIdentification()
- Replaced the code sample in the Getting Started (Example Quick Test) section
- Added Wrap Mode (VCR<1:0>) to the Volatile Configuration Register Bits table
- Added Dual I/O protocol (VECR<5>), VPP accelerator (VECR<3>), and Output driver strength <2:0> to the Volatile Enhanced Configuration Register Bits table

Rev. D – 05/11

- Rebranded as a technical note

Rev. C – 07/10

- Applied branding and formatting

Rev. B – 11/09

- Updated logos and branding

Rev. A – 04/09

- Initial Release

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