Technical Note

Software Drivers for M29W320DB and M29W320DT NOR Flash Memory

Introduction

This technical note provides library source code in C for the M29W320DB and M29W320DT parallel NOR Flash memory devices. The M29W320D has an array of 67 blocks: 4 Parameter Blocks and 63 Main Blocks. M29W320DT locates the Parameter Blocks at the top of the memory address space while the M29W320DB locates the Parameter Blocks starting from the bottom.

This application note supports the two devices, except where otherwise specified. There are two different M29W320D parts, the M29W320DT and M29W320DB, which will be referred to generically as the M29W320D.

Listings of the source code are located at the end of this document. The source code is also available in file form from www.micron.com or from your Micron distributor. The c1481_16.c and c1481_16.h files contain libraries for accessing the M29W320D Flash Memory in 16-bit bus mode, whereas the c1481_08.c and c1481_08.h files contain the 8-bit drivers for these devices. This is necessary because 8 bit and 16 modes have different command sequences and must be connected differently.

Also included in this technical note is an overview of the programming model for the M29W320D. This describes 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 by the user to compile and run. This document explains how the user should modify the source code for their individual target hardware. All source code is backed up by comments explaining how it is used and why it has been written as it has.

This application note does not replace the M29W320D data sheet. It refers to the data sheet throughout and it is necessary to have a copy in order to follow some of the explanations.

The software and accompanying documentation has been fully tested on a target platform. It is small in size and can be applied to any target hardware.

The AM29LV320D from AMD is software and hardware compatible with the M29W320D. Source code written to use the AMD AM29LV320D can be modified to use the Micron M29W320DB or M29W320DT instead.

M29W320D Programming Model

The M29W320D is a 32Mb (4Mb x8 or 2Mb x16) Flash memory device that can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is broken down into 67 blocks (4 parameter blocks and 63 main blocks) of varying sizes. Each block can be erased individually, or the whole chip can be erased at once, erasing all 32Mb.

The M29W320D is a single voltage device. The M29W320D is easy to use since the hardware does not need to cater for special bus signal levels. The voltages needed to erase the device are generated by charge pumps inside the device. Included in the device is a Program/Erase Controller. With first generation Flash Memory devices the software had to manually program all bytes to 00h before erasing to FFh using special programming sequences. The Program/Erase Controller in the M29W320D allows a simpler programming model to be used by taking care of all the 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.

The M29W320D does, however, require some high voltage bus signals if all of the functionality of the device is to be accessed. Each block can be protected against accidental programming or erasure. Protecting and unprotecting the blocks requires VID (about 12V) on some of the pins. Most applications of the device will not include these functions. However, blocks may be preprogramming, protected, and unprotected by an EPROM programmer prior to fitting into the hardware. Unprotected blocks may still be used to store data and parameters. By protecting a block, accidental data loss through software failure cannot occur.

Bus Operations and Commands

Most of the functionality of the M29W320D is 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, which modify the data stored or the behavior of the device. Only certain special sequences of WRITE operations are recognized as commands by the M29W320D. The various commands recognized by the M29W320D are listed in the Commands Table of the data sheet and can be grouped as follows:

- READ/RESET
- AUTO SELECT
- ERASE
- PROGRAM
- ERASE SUSPEND

The READ/RESET command returns the M29W320D to its reset state 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 AUTO SELECT command places the device in the auto select mode, which allows the user to read the electronic signature and block protection status of the device. The electronic signature (manufacturer and device codes) and the block protection status are accessed by reading different addresses while in auto select mode.

The ERASE commands are used to set all the bits to 1 in every memory location in the selected blocks (BLOCK ERASE command) or in the whole chip (CHIP ERASE command). All data previously stored in the erased blocks will be lost. The ERASE commands take longer to execute than the other commands, because entire blocks are erased at a time.

The PROGRAM command is used to modify the data stored at the specified address of the device. Note that programming cannot change bits from 0 to 1. It may therefore be necessary to erase the block before programming to addresses within it. Programming modifies a single word/byte at a time. Programming larger amounts of data must be done one word/byte at a time, by issuing a PROGRAM command, waiting for the command to complete, then issuing the next PROGRAM command, and so on. Each PROGRAM command requires 4 BUS WRITE operations to issue. However, after issuing the UNLOCK BYPASS command, PROGRAM commands only require two WRITE operations. Thus, using UNLOCK BYPASS saves some time when a large number of addresses need to be programmed at a time.

Issuing the ERASE SUSPEND command during a BLOCK ERASE operation will temporarily place the M29W320D in erase suspend mode. In this mode, the blocks not being erased may be read or programmed as if in the reset state of the device. This allows the user to access information stored in the device immediately rather than waiting until the BLOCK ERASE operation completes (typically 0.8s). The BLOCK ERASE operation is resumed when the device receives the ERASE RESUME command.

Status Register

While the M29W320D is programming or erasing, a READ from the device will output the status register of the Program/Erase Controller. This provides valuable information about the current PROGRAM or ERASE command. The status register bits are described in the Status Register Bits Table of the M29W320D data sheet. Their main use is to determine when programming or erasing is complete and whether it is successful or not. 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 Figure or the Data Toggle Flow Chart Figure in the data sheet. The library routines described in this technical 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) becoming 1 before the command has completed. 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 Commands Table of the M29W320D data sheet describes the sequences of bus WRITE operations that will be recognized by the Program/Erase Controller as valid commands. For example programming 9465h to the address 03E2h in 16-bit bus mode requires the user to write the following sequence (in C):

(unsigned int) (0x0555) = 0x00AA; *(unsigned int*) (0x02AA) = 0x0055; *(unsigned int*)(0x0555) = 0x00A0; *(unsigned int*)(0x03E2) = 0x9465; In 8-bit bus mode writing 65h to address 07C4h would require:

```
*(unsigned char*) (0x0AAA) =
0xAA; *(unsigned char*) (0x0555) =
0x55; *(unsigned char*) (0x0AAA) =
0xA0; *(unsigned char*) (0x07C4) = 0x65;
```

Note that due to the organization of the address bus and data bus, this example writes 65h to the same address for both 16-bit and 8-bit bus modes. The internal organization of the M29W320D is such that the LSB of the 16-bit mode is stored at the lower address of the 8bit mode. This example assumes that address 0000h of the M29W320D is mapped to address 0000h in the micro-processor address space. In practice it is likely that the Flash will have a base offset that needs to 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 on-going. Bit DQ7 will be the complement of the data being programmed.

There are only two possible outcomes to this programming command: success or failure. Success will be indicated by the toggle bit DQ6 no longer toggling but being constant at its programmed value (of 1 in our example) and the polling bit DQ7 also being at its programmed value (of 0 in our example). Failure will be indicated by the error bit DQ5 becoming 1 while the toggle bit DQ6 still toggles and the polling bit DQ7 remains the complement (1 in our example) of the data being programmed. Note that failure of the device itself is extremely unlikely. If the command fails it will normally be because the user is attempting to change a 0 to a 1 by programming. It is only possible to change a 0 to a 1 by erasing.

Writing C Code For The M29W320D

The low-level functions (drivers) described in this technical note have been provided to simplify the process of developing application code in C for Micron Flash memory. This enables users to concentrate on writing the high-level functions required for their particular applications. These high-level functions can access the Flash memory by calling the low-level drivers, hence keeping details of special command sequences away from the users' high level code. This results in source code both simpler and easier to maintain.

Code developed using the drivers provided can be decomposed 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 to communicate with the M29W320D. This implementation is hardware platform dependent as it is affected by which microprocessor the C code runs on and by where in the microprocessor's address space the memory device is located. The user must write the C functions appropriate to his hardware platform (see FlashRead() and FlashWrite() in the next section). 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 or 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 the user will access the memory device by calling the low-level functions.

By keeping the specific details of how to access the M29W320D away from the high-level functions, the user is left with code which is simple and easier to maintain. It also makes the user's high-level functions easier to apply to other Micron Flash memory devices.

When developing an application, the user is advised to proceed as follows:

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

C Library Functions Provided

The software library provided with this technical note provides the source code for the following functions:

- **FlashReadReset**() is used to reset the device into Read mode. Note that there should be no need call this function under normal operation as all of the other software library functions leave the devising this mode.
- **FlashAutoSelect**() is used to identify the manufacturer code, device code, and the block protection status of the device.
- **FlashBlockErase**() is used to erase one or more blocks in the device. Multiple blocks will be erased simultaneously to reduce the overall erase time. This function checks that none of the blocks specified reproductive and does not erase any blocks if some of the specified blocks are protected.
- **FlashChipErase**() is used to erase the entire chip. It will not erase any blocks if there is a protected block on the chip.
- **FlashProgram**() is used to program data arrays into the Flash. Only previously erased words/bytes can be programmed reliably. The function will not program any data if any of the words/bytes in the array fall inside a protected block.

The functions provided in the software library rely on the user implementing the hardware-specific bus operations and a suitable timing function. This is to be done by writing three functions:

- **FlashRead**() must be written to read a value from the Flash.
- **FlashWrite**() must be written to write a value to the Flash.
- **FlashPause**() must be written to provide a timer with microsecond resolution. This is used to wait while the Flash recovers from some conditions.

An example of these functions is provided in the source code. In many instances these functions can be written as macros and therefore will not incur the function call time overhead. The two functions which perform the basic I/O to the device have been provided for users who have awkward systems. For example, where the addressing system is peculiar or the data bus has D0..D7 of the device on D8..D15 of the microprocessor. They allow any user to quickly adapt the code to virtually any target system.

Throughout the functions assumptions have been made on the data types. These are:

- A char is 8 bits (1 byte). This is not the case in all microcontrollers. Where it is not it will be necessary to mask the unused bits of the word (particularly in the user's 8-bit mode FlashRead() function).
- An int is 16 bits (2 bytes). Again, like the char, if this is not the case it will be necessary to use a variable type which is 16 bits or longer and mask bits above 16 bits (for example in the user's 16-bit mode FlashRead() function).
- A long is 32 bits (4 bytes). It is necessary to have arithmetic greater than 16 bits in order to address the entire device.

Two approaches to the addressing are available: The desired address in the Flash can be specified by a 32-bit linear pointer or a 32-bit offset into the device could be provided by the user. The 16-bit bus mode FlashRead() functions in each case would declared as:

- unsigned int FlashRead(unsigned int *Addr);
- unsigned int FlashRead(unsigned long ulOff);

The pointer option has the advantage that it runs faster. The 32-bit offset must be changed to an address for each access and this involves 32 bit arithmetic. Using a 32-bit offset is, however, more portable because the resulting software can easily be changed to run on microprocessors with segmented memory spaces (such as the 8086). For maximum portability, all the functions in this technical note use a 32-bit unsigned long offset, rather than a pointer.

Porting the Drivers to the Target System

Before using the software in the target system, to do the following:

1. Define USE_M29W320DB or USE_M29W320DTdepending on whether an M29W320DB or M29W320DT is fitted. The top of the source file provided defines USE_M29W320DB as an example.
2. Write FlashRead(), FlashWrite(), and FlashPause() functions appropriate to the target hardware.
3. Search through the code for the /* DSI */ and /* ENI */ comments and disable/enable interrupts at the appropriate points.

The example FlashRead() and FlashWrite() functions provided in the source code should give the user a good idea of what is required and can be used in many instances without much modification. To test the source code in the target system, start by reading from the M29W320D. If it is erased, then only FFh data should be read. Next, read the manufacturer and device codes and check whether they are correct. If these functions work, then it is likely that all of the functions will work. However, they should all be tested thoroughly.

The programmer must take extra care when the device is accessed during an interrupt service routine. Three situations exist which must be considered:

1. When the device is in Read mode interrupts can freely read from the device.
2. Interrupts that do not access the device may be used during the PROGRAM, AUTOSELECT, and CHIP ERASE functions.
3. During the time critical section of the BLOCK ERASE function, interrupts are not permitted. An interrupt during this time may cause a time-out and result in some of the blocks not being erased correctly. The programmer should also take care when a RESET is applied during PROGRAM or ERASE operations. The Flash will be left in an indeterminate state and data could be lost.

C does not provide a standard library function for disabling interrupts. Furthermore, different applications have different tolerances on when interrupts may be disabled. Therefore no protection from the misuse of interrupts could be incorporated into the library source code. It is strongly recommended that the user disables interrupts where the /* DSI */ comments are placed in the source code. If this is not possible, then the user should erase one block at a time.

Limitations of the Software

The software provided does not implement a full set of the M29W320D's functionality. It is left to the user to implement the ERASE SUSPEND and UNLOCK BYPASS commands of the device. The standby mode is a hardware feature of the device and cannot be controlled through software.

Care should be taken in some of the while() loops. No time-outs have been implemented. Software execution may stop in one of the loops due to a hardware error. A /* TimeOut! */ comment has been put at these places and the user can add a timer to them to prevent the software failing.

The software only caters for one device in the system. To add software for more devices a mechanism for selecting the devices will be required. When an error occurs, the software simply returns the 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 M29W320DB and the M29W320DT single voltage NOR Flash memory 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.