Technical Note

M58WR0 64Mb Parallel NOR Flash
Memory Software Device Drivers

Introduction

This technical note provides a description of the library source code in C for the specified Parallel NOR Flash device. The c2525.c and the c2525.h files contain libraries for accessing the device.

This technical note does not duplicate or replace information from the specified Parallel NOR Flash data sheet. 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, explaining its use and the intent of its design.

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 low-level functions (drivers) described in this technical note are provided to simplify developing C application code for the device. The drivers enable developers to focus on writing the high-level functions required for their particular applications. The high-level functions access the device by calling the drivers that handle low-level command sequences. Thus, developer source code is both simpler and easier to maintain.

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

- Hardware-specific bus operations: Developer's applications are dependent on the microprocessor on which the code runs and where the device is mapped in the microprocessor's address space. In addition, code must be written according to the hardware platform.
- Low-level driver code: Drivers issue the correct sequences of WRITE operations for each command and interpret information received from the device during PROGRAM and ERASE operations. They also encode all details on how commands are issued to the device and how to interpret the device status register bits.
- High-level code: High level functions written by developers access the device by calling the drivers.

This software driver supports the Standard Software Driver Interface (STFL-I). To meet compatibility requirements, STFL-I assigns a number to each Flash memory block starting from 0 (address offset 0) up to the highest-address block number. For example, for a Flash device containing 64 blocks, STFL-I refers to the block at address offset 0 as block number 0, and to the last block, as block 63. Block numbers might be described differently in the data sheet.

Developers should proceed as follows when developing an application:

- Write a simple program to test the low level drivers and verify that they operate as expected on the target hardware and software environments.
- Write the application high-level code, which accesses the Flash device by calling the low level drivers.
- Test the completed application source code thoroughly.

Porting the Driver (User Change Area)

All 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 MT_uint8; (8 bits)
typedef char MT_sint8; (8 bits)
typedef unsigned short MT_uint16; (16 bits)
typedef short MT_sint16; (16 bits)
typedef unsigned int MT_uint32; (32 bits)
typedef int MT_sint32; (32 bits)
```

Device Type

Select the appropriate device #define statement:

```
#define USE_M58WR064KU

#define USE_M58WR064KL
```

Device Location

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

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

Flash Configuration

Select the appropriate Flash memory configuration to support a CPU board with either 1) an external 16 bit memory bus with one 16 bit Flash device, or 2) an external 32 bit memory bus with two 16 bit Flash devices:

```
#define USE_16BIT_CPU_ACCESSING_1_16BIT_FLASH

#define USE_32BIT_CPU_ACCESSING_2_16BIT_FLASH
```

Timeout

Timeouts are implemented in code loops to provide an exit to operations that otherwise would never terminate. Two options are possible:

If the compiler supports time.h, the following #define statement should be activated to prevent changes to the timeout setting because of the current evaluation hardware performance.

```
#define TIME_H_EXISTS
```

Without compiler support for time.h, timeouts should be set to a value that leads to a one 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: The value needs to be chosen according to the performance of the target hardware.

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.

Additional Considerations

The access timing data can sometimes be problematic. Therefore, in certain cases it may be necessary to change the functions FlashRead() and FlashWrite() when they are not compatible with the timing of the target hardware. These problems can be solved with a logic analyzer. The FlashRead() and FlashWrite() function examples provided in the source code should give developers what is required and useful, in many instances with little modification.

Developers must take extra care when the device is accessed during an interrupt service routine. In the source code there are some comments where the interrupts should be disabled or re-enabled.

C Library Functions

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

Table 2: Function Names and Descriptions

Function	Description
Flash()	Accesses all chip functions and acts as the main device interface. Function is guaranteed. Any functionality unsupported by the device can be detected and malfunctions can thus be avoided.
FlashBankErase()	Erases an entire Flash memory bank.
FlashBankReset()	Returns a bank to read array mode.
FlashBankResume()	Resumes a suspended operation on a bank.
FlashBankSuspend()	Suspends an operation on a bank.
FlashBlockErase()	Erases one or more blocks in the device.
FlashChipErase()	Erases the entire chip.
FlashProgram()	Programs one or more words to the array. The PROGRAM command cannot change a bit set to 0 back to 1.One of the ERASE commands must be used to set all the bits in a block or in the whole device from 0 to 1.
FlashSingleProgram()	Programs a single element.
FlashBlockLockDown()	Locks-down a block. Once locked-down, the block is locked and when Write Protect is LOW, the lock status of the block cannot be changed by using the software commands alone. The block will revert to the locked state when the device is reset or powered down.
FlashBlockProtect()	Locks (protects) a block in the Flash device. Once locked, the data in the block cannot be programmed or erased until the block is unlocked (unprotected).
FlashBlockUnprotect()	Unlocks (unprotects) a block in the Flash device. Once the block is unlocked, the data it contains can be erased or new data can be programmed to it.
FlashCheckBlockLockDownStatus()	Checks whether a block is locked-down.
FlashCheckBlockProtection()	Checks whether a block is locked.
FlashCheckCompatibility()	Checks the Flash device for compatibility
FlashChipUnprotect()	Unlocks all blocks in the Flash device. Once all the blocks are unlocked, the data contained in the blocks can be entirely erased or new data can be programmed.
FlashClearStatusRegister()	Clears the Status Register of a bank.
FlashErrorStr()	Generates a text string describing the detected error.
FlashReadCfi()	Reads the CFI data.
FlashReadDeviceID()	Reads the device codes of the Flash device.
FlashReadManufacturerCode()	Reads the manufacturer codes of the Flash device.
FlashReset()	Resets the device to read array mode. Note that there should be no need to call this function under normal operation as all the other software library functions leave the device in this mode.
FlashSetBurstConfig()	Writes a new value to the Flash memory configuration register. Attempting to set an invalid value generates an error and the command is ignored.

Software Driver Changes

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

Sample Code

The following sample code includes a main() function from which all Flash memory functions can be called and tested. Typical sample code begins by reading from the device; if the device is erased, FFFFh data should be output. Other device tests shown here include reading device ID information (device code, manufacturer code) to ensure it is correct and performing a BLOCK ERASE command. All other device functions should also be tested.

Table 3: Quick Device Test Sample Code

```
#include c2525.h

void main(void)
{
   ParameterType fp;                              /* contains all Flash parameters */
   ReturnType rRetVal;                            /* return Type enum */

   Flash(ReadManufacturerCode, &fp);
   printf ("Manufacturer Code: %04Xh\r\n",
      fp.ReadManufacturerCode.ucManufacturerCode);

   Flash(ReadDeviceid, &fp);
   printf ("Device Code: %04Xh\r\n",
      fp.ReadDeviceid.ucDeviceid);

   fp.BlockErase.ublBlockNr = 10;                 /* Block number 10 will be erased */
   rRetVal = Flash(BlockErase, &fp);              /* function execution */
}                                                 /* End function Main */
```

Software Limitations

The software described in this document does not implement all functionality of the 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

Parallel 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. A – 01/13

- 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