Technical Note

J3F 32Mb, 64Mb, 256Mb 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 c2105.c and c2105.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.

Note: To meet compatibility requirements, the software device driver numbers each block in a device starting from 0 (block 0 always has address offset 0) 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, it will always refer to the block with address offset 0 as block number 0, and to the last block as block number 63.

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.

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 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 UINT8; (8 bits)
typedef char SINT8; (8 bits)
typedef unsigned short UINT16; (16 bits)
typedef short SINT16; (16 bits)
typedef unsigned int UINT32; (32 bits)
typedef int SINT32; (32 bits)
```

Device Type

Use the appropriate define statement to choose the correct device.

```
#define PC28F256J3F95
#define PC28F320J3F75
#define PC28F640J3F75
```

Flash Memory 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 0, and must be adjusted appropriately.

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

Flash Configuration

Choose the correct device configuration.

```
#define USE_16BIT_CPU_ACCESSING_1_16BIT_FLASH
```

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.

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 the user a good idea of what is required and can be used in many instances with little modification.

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

Table 2: Function Names and Descriptions

Function	Description
Flash()	Accesses all device functions and acts as the main device interface. Available on all software drivers written in the driver format and should be used exclusively. Any functionality unsupported by the device can be detected and malfunctions can thus be avoided. The other functions are listed to offer a second-level interface when enhanced performance is required. Within the device driver, the functions are always used in the same way, and the function interface (names, return codes, parameters, and data types) remains unchanged, regardless of the device.
FlashBlockErase()	Erases a block in the device. A block cannot be erased when it is locked or V_{PP} is invalid (lower than V_{PPLK}). Attempting to do so generates an error.
FlashBlockLock()	Locks (protects) a block in the device. Once locked (protected), the data in the block cannot be programmed or erased until the block is unlocked (unprotected).
FlashBlockUnlock()	Unlocks a block in the device. Once the block is unlocked, the data it contains can be erased or new data can be programmed to it.
FlashCheckBlockProtection()	Checks whether a block is locked.
FlashCheckCompatibility()	Checks the device for compatibility.
FlashChipErase()	Erases the entire device. Locked blocks will not be erased. The device cannot be erased when V_{PP} is invalid (lower than V_{PPLK}). Attempting to do so generates an error.
FlashChipUnprotect()	Unlocks all blocks in the device. Once all the blocks are unlocked, the data contained in all the blocks can be entirely erased or new data can be programmed.
FlashClearStatusRegister()	Clears the status register.
FlashErrorStr()	Generates a text string describing the detected error.
FlashProgram()	Programs data arrays to the device. Only previously erased elements can be programmed reliably. Locked blocks cannot be programmed, and PROGRAM operations cannot be performed when V_{PP} is invalid.
FlashProtectionRegisterProgram()	Programs the protection register.
FlashReadCfi()	Checks whether if the CFI is supported and then read the CFI data at the specified offset.
FlashReadDeviceId()	Reads the device codes.
FlashReadManufacturerCode()	Reads the manufacturer codes.
FlashReadProtectionRegister()	Reads a location in the protection register.
FlashReadProtectRegisterProgram()	Reads the status register.
FlashReset()	Resets the device to read array mode. All the other software library functions leave the device in this mode under normal operation; therefore, function does not need to be called out.
FlashSingleProgram()	Programs a single element.

Table 2: Function Names and Descriptions (Continued)

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 BlockErase command. All other device functions should also be tested.

Table 3: Quick Device Test Sample Code

```
#include "c2105.h"

void main(void)
{
   ParameterType flashPara;                    /* parameters used for all opera-
tions */
   ReturnType retVal;                          /* return Type enum */

   Flash(ReadManufacturerCode, &flashPara);
   printf ("Manufacturer Code: %08xh/r/n",
      flashPara.ReadManufacturerCode.ucManufacturerCode);

   Flash(ReadDeviceID, &flashPara);
   printf ("Device Code: %08xh/r/n",
      flashPara.ReadDeviceID.ucDeviceID);

   flashPara.BlockErase.ublBlockNr = 10;       /* designates block 10 to be erased
*/
   rRetVal = Flash(BlockErase, &flashPara);    /* function executed; block 10
erased */
}                                              /* 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