AN1199
APPLICATION NOTE

Software Drivers for the M29F016B Flash Memory

CONTENTS

- INTRODUCTION
- THE M29F016B PROGRAMMING MODEL
- WRITING C CODE FOR THE M29F016B
- C LIBRARY FUNCTIONS PROVIDED
- PORTING THE DRIVERS TO THE TARGET SYSTEM
- LIMITATIONS OF THE SOFTWARE
- CONCLUSION
- c1199_08.h LISTING
- c1199_08.c LISTING

INTRODUCTION

This application note provides library source code in C for the M29F016B Flash Memory. The M29F016B has a rated power supply of 5V.

Listings of the source code can be found at the end of this document. The source code is also available in file form from the internet site http://www.st.com or from your STMicroelectronics distributor. The c1199_08.c and c1199_08.h files contain libraries for accessing the M29F016B Flash Memory.

Also included in this application note is an overview of the programming model for the M29F016B. 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 by the user in order to compile and run. The application note explains how the user should modify the source code for their individual target hardware. All of the 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 M29F016B 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 Am29F016B from AMD is software and hardware compatible with the M29F016B. Source code written to use AMD's Am29F016B can easily be modified to use STMicroelectronics's M29F016B instead (see STMicroelectronics application note AN1185 for more details).

THE M29F016B PROGRAMMING MODEL

The M29F016B is a 16Mb (2Mb x8) Flash Memory which can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is broken down into 32 blocks, each 64 Kbytes in size. Each block can be erased individually, or the whole chip can be erased at once, erasing all 16Mb.

The M29F016B is a single voltage device. It differs from first generation devices which require a 12V supply to program or erase. The M29F016B is therefore easier 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 of the bytes to 00h before erasing to FFh using special programming sequences. The Program/Erase Controller in the M29F016B 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 M29F016B does, however, require some high voltage bus signals if all of the functionality of the device is to be accessed. Blocks can be protected in groups 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 pre-programmed, 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 group of blocks, accidental data loss through software failure cannot occur.

Bus Operations and Commands

Most of the functionality of the M29F016B 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 behaviour of the device. Only certain special sequences of write operations are recognized as commands by the M29F016B. The various commands recognized by the M29F016B are listed in the Commands Table of the datasheet and can be grouped as follows:

1. Read/Reset
2. Auto Select
3. Erase
4. Program
5. Erase Suspend

The Read/Reset command returns the M29F016B 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 whilst in the 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 byte at a time. Programming larger amounts of data must be done one 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 write operations to issue. However, after issuing the Unlock Bypass command, Program commands only require 2 write operations. Using Unlock Bypass will thus save 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 M29F016B 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 0.6s for the M29F016B. The Block Erase operation is resumed when the device receives the Erase Resume command.

The Status Register

While the M29F016B 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 M29F016B 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 datasheet. 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) 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.

A Detailed Example

The Commands Table of the M29F016B Data Sheet describes the sequences of Byte write operations that will be recognized by the Program/Erase Controller as valid commands. For example programming 65h to the address 03E2h requires the user to write the following sequence (in C):

```
*(unsigned char*)(0x0555) = 0xAA;
*(unsigned char*)(0x02AA) = 0x55;
*(unsigned char*)(0x0555) = 0xA0;
*(unsigned char*)(0x03E2) = 0x65;
```

This example assumes that address 0000h of the M29F016B is mapped to address 0000h in the microprocessor address space. In practice it is likely that the Flash will have a base offset which 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 M29F016B

The low-level functions (drivers) described in this application note have been provided to simplify the process of developing application code in C for the STMicroelectronics Flash Memories (M29F016B). This enables users to concentrate on writing the high level functions required for their particular applications. These high level functions can access the Flash Memories by calling the low level drivers, hence keeping details of special command sequences away from the users' high level code: this will result in source code both simpler and easier to maintain.

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

1. the hardware specific bus operations
2. the low-level drivers
3. the 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 M29F016B. 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 will have to 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 M29F016B 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 STMIcroelectronics Flash Memories.

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

– first 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.
– then the user should write the high level code for his application, which will access the Flash Memories by calling the low level drivers provided.
– finally test the complete application source code thoroughly.

C LIBRARY FUNCTIONS PROVIDED

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

`FlashReadReset()` 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 of the other software library functions leave the device in 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 are protected 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 group of blocks on the chip.

`FlashProgram()` is used to program data arrays into the Flash. Only previously erased bytes can be programmed reliably. The function will not program any data if any of the 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 as well as a suitable timing function. This is to be done by writing three functions as follows:

– **`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 **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.

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 **FlashRead()** functions in each case would declared as:

```
unsigned char FlashRead( unsigned char *Addr);
unsigned char FlashRead( unsigned long ulOff);
```

The pointer option has the advantage that it runs faster. The 32 bit offset needs to be changed to an address for each access and this involves 32 bit arithmetic. Using a 32 bit offset is, however, more portable since 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 application 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 the user needs to do the following:

1. Write **FlashRead()**, **FlashWrite()** and **FlashPause()** functions appropriate to the Target Hardware.
2. 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 simply reading from the M29F016B. If it is erased then only FFh data should be read. Next read the Manufacturer and Device codes and check they are correct. If these functions work then it is likely that all of the functions will work but they should all be tested thoroughly.

The programmer needs to 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 which 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 M29F016B'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 M29F016B single voltage Flash Memory is an ideal product for embedded and other computer systems, able to be easily interfaced to microprocessors and driven with simple software drivers written in the C language.

```
/****c1199_08.h*Header File for c1199_08.c************************************

   Filename:    c1199_08.h
   Description: Header file for c1199_08.c. Consult the C file for details

   Copyright (c) 1999 STMicroelectronics.

   THIS PROGRAM IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND,EITHER
   EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY
   OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK
   AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
   PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
   REPAIR OR CORRECTION.
******************************************************************************/


/******************************************************************************
Commands for the various functions
******************************************************************************/
#define FLASH_READ_MANUFACTURER    (-2)
#define FLASH_READ_DEVICE_CODE     (-1)

/******************************************************************************
Error Conditions and return values.

See end of C file for explanations and help
******************************************************************************/
#define FLASH_BLOCK_PROTECTED      (0x01)
#define FLASH_BLOCK_UNPROTECTED    (0x00)
#define FLASH_BLOCK_NOT_ERASED     (0xFF)
#define FLASH_BLOCK_ERASE_FAILURE  (0xFE)
#define FLASH_BLOCK_ERASED         (0xFD)

#define FLASH_SUCCESS              (-1)
#define FLASH_POLL_FAIL            (-2)
#define FLASH_TOO_MANY_BLOCKS      (-3)
#define FLASH_MPU_TOO_SLOW         (-4)
#define FLASH_BLOCK_INVALID        (-5)
#define FLASH_PROGRAM_FAIL         (-6)
#define FLASH_OFFSET_OUT_OF_RANGE  (-7)
#define FLASH_WRONG_TYPE           (-8)
#define FLASH_BLOCK_FAILED_ERASE   (-9)
#define FLASH_ERASE_FAIL           (-14)
#define FLASH_TOGGLE_FAIL          (-15)


/******************************************************************************
Function Prototypes
******************************************************************************/
extern unsigned char FlashRead( unsigned long ulOff );
extern void FlashReadReset( void );
extern int FlashAutoSelect( int iFunc );
extern int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[]);
extern int FlashChipErase( unsigned char *ucResults );
extern int FlashProgram( unsigned long ulOff, size_t NumBytes, void *Array );
extern char *FlashErrorStr( int iErrNum );
```

```
/****c1199_08.c*16Mb Flash Memory*********************************************

   Filename:    c1199_08.c
   Description: Library routines for the M29F016B 16Mb (2Mb x8) Flash Memory.

   Revision:    1.00
   Date:        29/09/99
   Author:      Alex Nairac, Oxford Technical Solutions (www.ots.ndirect.co.uk)
   Copyright (c) 1999 STMicroelectronics.

   THIS PROGRAM IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER
   EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY
   OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK
   AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
   PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
   REPAIR OR CORRECTION.

******************************************************************************

   Version History.

   Ver.   Date      Comments

   1.00   29/09/99  Initial Release of the Software.

******************************************************************************

   This source file provides library C code for using the M29F016B device.

   The following functions are available in this library:

      FlashReadReset()   to reset the flash for normal memory access
      FlashAutoSelect()  to get information about the device
      FlashBlockErase()  to erase one or more blocks
      FlashChipErase()   to erase the whole chip
      FlashProgram()     to program a byte or an array
      FlashErrorStr()    to return the error string of an error

   For further information consult the Data Sheet and the Application Note. The
   Application Note gives information about how to modify this code for a
   specific application.

   The hardware specific functions which need to be modified by the user are:

      FlashWrite() for writing a byte to the flash
      FlashRead()  for reading a byte from the flash
      FlashPause() for timing short pauses (in micro seconds)

   A list of the error conditions is given at the end of the code.

   There are no timeouts implemented in the loops in the code. At each point
   where an infinite loop is implemented a comment /# TimeOut! #/ has been
   placed. It is up to the user to implement these to avoid the code hanging
   instead of timing out.

   Since C does not include a method for disabling interrupts to keep time-
   critical sections of code from being disabled. The user may wish to disable
   interrupt during parts of the code to avoid the FLASH_MPU_TOO_SLOW error from
   occuring if an interrupt occurs at the wrong time. Where interrupt should be
```

```
   disabled and re-enabled there is a /# DSI! #/ or /# ENI! #/ comment.

   The source code assumes that the compiler implements the numerical types as

   unsigned char     8 bits
   unsigned int     16 bits
   unsigned long    32 bits

   Additional changes to the code will be necessary if these are not correct.

*******************************************************************************/

#include <stdlib.h>

#include “c1199_08.h”        /* Header file with global prototypes */


/*******************************************************************************
Constants
*******************************************************************************/
#define COUNTS_PER_MICROSECOND (200)
#define MANUFACTURER_ST (0x20) /* Manufacturer code */
#define BASE_ADDR ((volatile unsigned char*)0x0000)
   /* BASE_ADDR is the base address of the flash, see the functions FlashRead()
      and FlashWrite(). Some applications which require a more complicated
      FlashRead() or FlashWrite() may not use BASE_ADDR */
#define ANY_ADDR (0x0000L)
   /* Any address offset within the Flash Memory will do */

#define EXPECTED_DEVICE (0xAD) /* Device code for the M29F016B */

static const unsigned long BlockOffset[] = /* Offset from BASE_ADDR of blocks */
{
   0x000000L,  /* Start offset of block 0 */
   0x010000L,  /* Start offset of block 1 */
   0x020000L,  /* Start offset of block 2 */
   0x030000L,  /* Start offset of block 3 */
   0x040000L,  /* Start offset of block 4 */
   0x050000L,  /* Start offset of block 5 */
   0x060000L,  /* Start offset of block 6 */
   0x070000L,  /* Start offset of block 7 */
   0x080000L,  /* Start offset of block 8 */
   0x090000L,  /* Start offset of block 9 */
   0x0A0000L,  /* Start offset of block 10 */
   0x0B0000L,  /* Start offset of block 11 */
   0x0C0000L,  /* Start offset of block 12 */
   0x0D0000L,  /* Start offset of block 13 */
   0x0E0000L,  /* Start offset of block 14 */
   0x0F0000L,  /* Start offset of block 15 */
   0x100000L,  /* Start offset of block 16 */
   0x110000L,  /* Start offset of block 17 */
   0x120000L,  /* Start offset of block 18 */
   0x130000L,  /* Start offset of block 19 */
   0x140000L,  /* Start offset of block 20 */
   0x150000L,  /* Start offset of block 21 */
   0x160000L,  /* Start offset of block 22 */
   0x170000L,  /* Start offset of block 23 */
   0x180000L,  /* Start offset of block 24 */
   0x190000L,  /* Start offset of block 25 */
```

```
   0x1A0000L,  /* Start offset of block 26 */
   0x1B0000L,  /* Start offset of block 27 */
   0x1C0000L,  /* Start offset of block 28 */
   0x1D0000L,  /* Start offset of block 29 */
   0x1E0000L,  /* Start offset of block 30 */
   0x1F0000L   /* Start offset of block 31 */
};

#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))
#define FLASH_SIZE (0x200000L) /* 2M */


/*****************************************************************************
Static Prototypes

The following functions are only needed in this module.
*****************************************************************************/
static unsigned char FlashWrite( unsigned long ulOff, unsigned char ucVal );
static void FlashPause( unsigned int uMicroSeconds );
static int FlashDataToggle( void );
static int FlashBlockFailedErase( unsigned char ucBlock );


/*****************************************************************************
The function FlashDataPoll() declared below is not used by this library but is
provided as an illustration of the Data Polling Flow Chart
*****************************************************************************/
#define ILLUSTRATION_ONLY
#ifndef ILLUSTRATION_ONLY
static int FlashDataPoll( unsigned long ulOff, unsigned char ucVal );
#endif /* !ILLUSTRATION_ONLY */


/*****************************************************************************
Function:    unsigned char FlashWrite( unsigned long ulOff, unsigned char ucVal)
Arguments:   ulOff is the byte offset in the flash to write to
   ucVal is the value to be written
Returns:     ucVal
Description: This function is used to write a byte to the flash. On many
   microprocessor systems a macro can be used instead, increasing the speed of
   the flash routines. For example:

#define FlashWrite( ulOff, ucVal ) ( BASE_ADDR[ulOff] = (unsigned char) ucVal )

   A function is used here instead to allow the user to expand it if necessary.
   The function is made to return ucVal so that it is compatible with the macro.

Pseudo Code:
   Step 1: Write ucVal to the byte offset in the flash
   Step 2: return ucVal
*****************************************************************************/
static unsigned char FlashWrite( unsigned long ulOff, unsigned char ucVal )
{
   /* Step1, 2: Write ucVal to the byte offset in the flash and return it */
   return BASE_ADDR[ulOff] = ucVal;
}


/*****************************************************************************
```

```
Function:    unsigned char FlashRead( unsigned long ulOff )
Arguments:   ulOff is the byte offset into the flash to read from
Returns:     The unsigned char at the byte offset
Description: This function is used to read a byte from the flash. On many
   microprocessor systems a macro can be used instead, increasing the speed of
   the flash routines. For example:

#define FlashRead( ulOff ) ( BASE_ADDR[ulOff] )

  A function is used here instead to allow the user to expand it if necessary.

Pseudo Code:
  Step 1: Return the value at byte offset ulOff
*******************************************************************************/
unsigned char FlashRead( unsigned long ulOff )
{
   /* Step 1 Return the value at byte offset ulOff */
   return BASE_ADDR[ulOff];
}


/*******************************************************************************
Function:    void FlashPause( unsigned int uMicroSeconds )
Arguments:   uMicroSeconds is the length of the pause in microseconds
Returns:     none
Description: This routine returns after uMicroSeconds have elapsed. It is used
   in several parts of the code to generate a pause required for correct
   operation of the flash part.

   The routine here works by counting. The user may already have a more suitable
   routine for timing which can be used.

Pseudo Code:
   Step 1: Compute count size for pause.
   Step 2: Count to the required size.
*******************************************************************/
static void FlashPause( unsigned int uMicroSeconds )
{
   volatile unsigned long ulCountSize;

   /* Step 1: Compute the count size */
   ulCountSize = (unsigned long)uMicroSeconds * COUNTS_PER_MICROSECOND;

   /* Step 2: Count to the required size */
   while( ulCountSize > 0 )   /* Test to see if finished */
      ulCountSize--;          /* and count down */
}


/*******************************************************************************
Function:      void FlashReadReset( void )
Arguments:     none
Return Value:  none
Description:   This function places the flash in the Read mode described
   in the Data Sheet. In this mode the flash can be read as normal memory.

   All of the other functions leave the flash in the Read mode so this is
   not strictly necessary. It is provided for completeness.
```

```
Note: A wait of 10us is required if the command is called during a program or
   erase instruction. This is included here to guarantee correct operation. The
   functions in this library call this function if they suspect an error
   during programming or erasing so that the 10us pause is included. Otherwise
   they use the single instruction technique for increased speed.

Pseudo Code:
   Step 1: write command sequence (see Commands Table of the Data Sheet)
   Step 2: wait 10us
*******************************************************************************/
void FlashReadReset( void )
{
   /* Step 1: write command sequence */
   FlashWrite( 0x0555L, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x02AAL, 0x55 );  /* 2nd Cycle */
   FlashWrite( ANY_ADDR, 0xF0 ); /* 3rd Cycle: write 0xF0 to ANY address */

   /* Step 2: wait 10us */
   FlashPause( 10 );
}


/*******************************************************************************
Function:      int FlashAutoSelect( int iFunc )
Arguments:     iFunc should be set either to the Read Signature values or to the
   block number. The header file defines the values for reading the Signature.
   Note: the first block is Block 0

Return Value:  When iFunc is >= 0 the function returns FLASH_BLOCK_PROTECTED
   (01h) if the block is protected and FLASH_BLOCK_UNPROTECTED (00h) if it is
   unprotected. See the Auto Select command in the Data Sheet for further
   information.

   When iFunc is FLASH_READ_MANUFACTURER (-2) the function returns the
   manufacturer's code. The Manufacturer code for ST is 20h.

   When iFunc is FLASH_READ_DEVICE_CODE (-1) the function returns the Device
   Code.  The device code for the M29F016B is ADh.

   When iFunc is invalid the function returns FLASH_BLOCK_INVALID (-5)

Description:   This function can be used to read the electronic signature of the
   device, the manufacturer code or the protection level of a block.

Pseudo Code:
   Step 1:  Send the Auto Select command to the device
   Step 2:  Read the required function from the device.
   Step 3:  Return the device to Read mode.
*******************************************************************************/
int FlashAutoSelect( int iFunc )
{
   int iRetVal; /* Holds the return value */

   /* Step 1: Send the Auto Select command */
   FlashWrite( 0x0555L, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x02AAL, 0x55 );  /* 2nd Cycle */
   FlashWrite( 0x0555L, 0x90 );  /* 3rd Cycle */

   /* Step 2: Read the required function */
```

```
   if( iFunc == FLASH_READ_MANUFACTURER )
      iRetVal = (int) FlashRead( 0x0000L ); /* A0 = A1 = 0 */

   else if( iFunc == FLASH_READ_DEVICE_CODE )
      iRetVal = (int) FlashRead( 0x0001L ); /* A0 = 1, A1 = 0 */

   else if( (iFunc >= 0) && (iFunc < NUM_BLOCKS) )
      iRetVal = FlashRead( BlockOffset[iFunc] + 0x0002L );
                                          /* A0 = 0, A1 = 1 */
   else
      iRetVal = FLASH_BLOCK_INVALID;

   /* Step 3: Return to Read mode */
   FlashWrite( ANY_ADDR, 0xF0 ); /* Use single instruction cycle method */

   return iRetVal;
}


/*****************************************************************************
Function:     int FlashBlockErase( unsigned char ucNumBlocks,
   unsigned char ucBlock[]  )
Arguments:    ucNumBlocks holds the number of blocks in the array ucBlock
   ucBlock is an array containing the blocks to be erased.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS             (-1)
   FLASH_TOO_MANY_BLOCKS     (-3)
   FLASH_MPU_TOO_SLOW        (-4)
   FLASH_WRONG_TYPE          (-8)
   FLASH_ERASE_FAIL          (-14)
   Number of the first protected or invalid block

      The user's array, ucBlock[] is used to report errors on the specified
   blocks. If a time-out occurs because the MPU is too slow then the blocks
   in ucBlocks which are not erased are overwritten with FLASH_BLOCK_NOT_ERASED
   (FFh) and the function returns FLASH_MPU_TOO_SLOW.
      If an error occurs during the erasing of the blocks the function returns
   FLASH_ERASE_FAIL.
      If both errors occur then the function will set the ucBlock array to
   FLASH_BLOCK_NOT_ERASED for the unerased blocks. It will return
   FLASH_ERASE_FAIL even though the FLASH_MPU_TOO_SLOW has also occurred.

Description:  This function erases up to ucNumBlocks in the flash. The blocks
   can be listed in any order. The function does not return until the blocks are
   erased. If any blocks are protected or invalid none of the blocks are erased.

   During the Erase Cycle the Data Toggle Flow Chart of the Data Sheet is
   followed. The polling bit, DQ7, is not used.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check for protected or invalid blocks
   Step 3:  Write Block Erase command
   Step 4:  Check for time-out blocks
   Step 5:  Wait for the timer bit to be set.
   Step 6:  Follow Data Toggle Flow Chart until Program/Erase Controller has
            completed
   Step 7:  Return to Read mode (if an error occurred)
*****************************************************************************/
```

```
int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[] )
{
   unsigned char ucCurBlock;    /* Range Variable to track current block */
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
   unsigned char ucFirstRead, ucSecondRead; /* used to check toggle bit DQ2 */

   /* Step 1: Check for correct flash type */
   if( !(FlashAutoSelect( FLASH_READ_MANUFACTURER ) == MANUFACTURER_ST)
   ||  !(FlashAutoSelect( FLASH_READ_DEVICE_CODE ) == EXPECTED_DEVICE ) )
      return FLASH_WRONG_TYPE;

   /* Step 2: Check for protected or invalid blocks. */
   if( ucNumBlocks > NUM_BLOCKS )     /* Check specified blocks <= NUM_BLOCKS */
      return FLASH_TOO_MANY_BLOCKS;

   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      /* Use FlashAutoSelect to find protected or invalid blocks*/
      if( FlashAutoSelect((int)ucBlock[ucCurBlock]) != FLASH_BLOCK_UNPROTECTED )
         return (int)ucBlock[ucCurBlock];  /* Return protected/invalid blocks */
   }

   /* Step 3: Write Block Erase command */
   FlashWrite( 0x0555L, 0xAA );
   FlashWrite( 0x02AAL, 0x55 );
   FlashWrite( 0x0555L, 0x80 );
   FlashWrite( 0x0555L, 0xAA );
   FlashWrite( 0x02AAL, 0x55 );
   /* DSI!: Time critical section. Additional blocks must be added every 50us */
   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      FlashWrite( BlockOffset[ucBlock[ucCurBlock]], 0x30 );

      /* Check for Erase Timeout Period (is bit DQ3 set?) */
      if( (FlashRead( BlockOffset[ucBlock[0]] ) & 0x08) == 0x08 )
         break; /* Cannot set any more blocks due to timeout */
   }
   /* ENI! */

   /* Step 4: Check for time-out blocks */
   /* if timeout occurred then check if current block is erasing or not */
   /* Use DQ2 of status register, toggle implies block is erasing */
   if( ucCurBlock < ucNumBlocks )
   {
      ucFirstRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x04;
      ucSecondRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x04;
      if( ucFirstRead != ucSecondRead )
      {
         ucCurBlock++; /* Point to the next block */
      }

      if( ucCurBlock < ucNumBlocks )
      {
         /* Indicate that some blocks have been timed out of the erase list */
         iRetVal = FLASH_MPU_TOO_SLOW;
      }

      /* Now specify all other blocks as not being erased */
      while( ucCurBlock < ucNumBlocks )
```

```
      {
         ucBlock[ucCurBlock++] = FLASH_BLOCK_NOT_ERASED;
      }
   }

   /* Step 5: Wait for the Erase Timer Bit (DQ3) to be set */
   while( 1 )  /* TimeOut!: If, for some reason, the hardware fails then this
               loop may not exit. Use a timer function to implement a timeout
               from the loop. */
   {
      if( ( FlashRead( BlockOffset[ucBlock[0]] ) & 0x08 ) == 0x08 )
         break; /* Break when device starts the erase cycle */
   }

   /* Step 6: Follow Data Toggle Flow Chart until Program/Erase Controller
              completes */
   if( FlashDataToggle() != FLASH_SUCCESS )
   {
      iRetVal = FLASH_ERASE_FAIL;
      /* Step 7: Return to Read mode (if an error occurred) */
      FlashReadReset();
   }

   return iRetVal;
}


/*****************************************************************************
Function:     int FlashChipErase( unsigned char *ucResults )
Arguments:    ucResults is a pointer to an array where the results will be
   stored. If ucResults == NULL then no results are stored. Otherwise the
   results are written to the array if an error occurs. The array is left
   unchanged if the function returns FLASH_SUCCESS.
   The errors written to the array are:
      FLASH_BLOCK_ERASED (FDh)        if the block erased correctly
      FLASH_BLOCK_ERASE_FAILURE (FEh) if the block failed to erased
Return Value: On success the function returns FLASH_SUCCESS (-1)
   If a block is protected then the function returns the number of the block and
   no blocks are erased.
   If the erase algorithms fails then the function returns FLASH_ERASE_FAIL (-2)
   If the wrong type of flash is detected then FLASH_WRONG_TYPE (-8) is
   returned.
Description: The function can be used to erase the whole flash chip so long as
   no groups of blocks are protected. If any groups of blocks are protected then
   nothing is erased.

Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: Check that all blocks are unprotected
   Step 3: Send Chip Erase Command
   Step 4: Follow Data Toggle Flow Chart until Program/Erase Controller has
           completed.
   Step 5: Check for blocks erased correctly
   Step 6: Return to Read mode (if an error occurred)
*****************************************************************************/
int FlashChipErase( unsigned char *ucResults )
{
   unsigned char ucCurBlock; /* Used to track the current block in a range */
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
```

```
   /* Step 1: Check for correct flash type */
   if( !(FlashAutoSelect( FLASH_READ_MANUFACTURER ) == MANUFACTURER_ST)
   ||  !(FlashAutoSelect( FLASH_READ_DEVICE_CODE ) == EXPECTED_DEVICE ) )
      return FLASH_WRONG_TYPE;

   /* Step 2: Check that all blocks are unprotected */
   for( ucCurBlock = 0; ucCurBlock < NUM_BLOCKS; ucCurBlock++ )
   {
      if( FlashAutoSelect( (int)ucCurBlock ) != FLASH_BLOCK_UNPROTECTED )
         return (int)ucCurBlock;  /* Return the first protected block */
   }

   /* Step 3: Send Chip Erase Command */
   FlashWrite( 0x0555L, 0xAA );
   FlashWrite( 0x02AAL, 0x55 );
   FlashWrite( 0x0555L, 0x80 );
   FlashWrite( 0x0555L, 0xAA );
   FlashWrite( 0x02AAL, 0x55 );
   FlashWrite( 0x0555L, 0x10 );

   /* Step 4: Follow Data Toggle Flow Chart until Program/Erase Controller has
              completed */
   if( FlashDataToggle() != FLASH_SUCCESS )
   {
      iRetVal = FLASH_ERASE_FAIL;
   }

   /* Step 5: Check for blocks erased correctly */
   if( iRetVal != FLASH_SUCCESS && ucResults != NULL )
   {
      for( ucCurBlock = 0; ucCurBlock < NUM_BLOCKS; ucCurBlock++ )
      {
         if( FlashBlockFailedErase( ucCurBlock )
             == FLASH_BLOCK_FAILED_ERASE )
            ucResults[ucCurBlock] = FLASH_BLOCK_ERASE_FAILURE;
        else
            ucResults[ucCurBlock] = FLASH_BLOCK_ERASED;
      }
   }

   /* Step 6: Return to Read mode (if an error occurred) */
   if( iRetVal != FLASH_SUCCESS )
      FlashReadReset();

   return iRetVal;
}


/*****************************************************************************
Function:     int FlashProgram( unsigned long ulOff, size_t NumBytes,
   void *Array )
Arguments:    ulOff is the byte offset into the flash to be programmed
   NumBytes holds the number of bytes in the array.
   Array is a pointer to the array to be programmed.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS              (-1)
   FLASH_PROGRAM_FAIL         (-6)
   FLASH_OFFSET_OUT_OF_RANGE  (-7)
```

```
   FLASH_WRONG_TYPE           (-8)
   Number of the first protected or invalid block

   On success the function returns FLASH_SUCCESS (-1).
   The function returns FLASH_PROGRAM_FAIL (-6) if a programming failure occurs.
   If the address range to be programmed exceeds the address range of the Flash
   Device the function returns FLASH_OFFSET_OUT_OF_RANGE (-7) and nothing is
   programmed.
   If the wrong type of flash is detected then FLASH_WRONG_TYPE (-8) is
   returned and nothing is programmed.
   If part of the address range to be programmed falls within a protected block,
   the function returns the number of the first protected block encountered and
   nothing is programmed.
Description: This function is used to program an array into the flash. It does
   not erase the flash first and may fail if the block(s) are not erased first.

Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: Check the offset range is valid
   Step 3: Check that the block(s) to be programmed are not protected
   Step 4: While there is more to be programmed
   Step 5:  Check for changes from '0' to '1'
   Step 6:  Program the next byte
   Step 7:  Follow Data Toggle Flow Chart until Program/Erase Controller has
            completed
   Step 8:  Return to Read Mode (if an error occurred)
   Step 9:  Update pointers
*****************************************************************************/
int FlashProgram( unsigned long ulOff, size_t NumBytes, void *Array )
{
   unsigned char *ucArrayPointer; /* Use an unsigned char to access the array */
   unsigned long ulLastOff;       /* Holds the last offset to be programmed */
   unsigned char ucCurBlock;      /* Range Variable to track current block */

   /* Step 1: Check for correct flash type */
   if( !(FlashAutoSelect( FLASH_READ_MANUFACTURER ) == MANUFACTURER_ST)
   ||  !(FlashAutoSelect( FLASH_READ_DEVICE_CODE ) == EXPECTED_DEVICE ) )
      return FLASH_WRONG_TYPE;

   /* Step 2: Check the offset and range are valid */
   ulLastOff = ulOff+NumBytes-1;
   if( ulLastOff >= FLASH_SIZE )
      return FLASH_OFFSET_OUT_OF_RANGE;

   /* Step 3: Check that the block(s) to be programmed are not protected */
   for( ucCurBlock = 0; ucCurBlock < NUM_BLOCKS; ucCurBlock++ )
   {
      /* If the address range to be programmed ends before this block */
      if( BlockOffset[ucCurBlock] > ulLastOff )
         break;                   /* then we are done */
      /* Else if the address range starts beyond this block */
      else if( (ucCurBlock < (NUM_BLOCKS-1)) &&
               (ulOff >= BlockOffset[ucCurBlock+1]) )
         continue;                /* then skip this block */
      /* Otherwise if this block is not unprotected */
      else if( FlashAutoSelect((int)ucCurBlock) != FLASH_BLOCK_UNPROTECTED )
         return (int)ucCurBlock;  /* Return first protected block */
   }
```

```
   /* Step 4: While there is more to be programmed */
   ucArrayPointer = (unsigned char *)Array;
   while( ulOff <= ulLastOff )
   {
      /* Step 5: Check for changes from '0' to '1' */
      if( ~FlashRead( ulOff ) & *ucArrayPointer )
         /* Indicate failure as it is not possible to change a '0' to a '1'
            using a Program command. This must be done using an Erase command */
         return FLASH_PROGRAM_FAIL;

      /* Step 6: Program the next byte */
      FlashWrite( 0x0555L, 0xAA );  /* 1st cycle */
      FlashWrite( 0x02AAL, 0x55 );  /* 2nd cycle */
      FlashWrite( 0x0555L, 0xA0 );  /* Program command */
      FlashWrite( ulOff, *ucArrayPointer );   /* Program value */

      /* Step 7: Follow Data Toggle Flow Chart until Program/Erase Controller
                 has completed */
      /* See Data Toggle Flow Chart of the Data Sheet */
      if( FlashDataToggle() == FLASH_TOGGLE_FAIL )
      {
         /* Step 8: Return to Read Mode (if an error occurred) */
         FlashReadReset();
         return FLASH_PROGRAM_FAIL;
      }

      /* Step 9: Update pointers */
      ulOff++;
      ucArrayPointer++;
   }

   return FLASH_SUCCESS;
}


/*****************************************************************************
Function:     static int FlashDataToggle( void )
Arguments:    none
Return Value: The function returns FLASH_SUCCESS if the Program/Erase Controller
   is successful or FLASH_TOGGLE_FAIL if there is a problem.
Description: The function is used to monitor the Program/Erase Controller during
   erase or program operations. It returns when the Program/Erase Controller has
   completed. In the M29F016B Data Sheet the Data Toggle Flow Chart shows the
   operation of the function.

Pseudo Code:
   Step 1: Read DQ6 (into a byte)
   Step 2: Read DQ5 and DQ6 (into another byte)
   Step 3: If DQ6 did not toggle between the two reads then return FLASH_SUCCESS
   Step 4: Else if DQ5 is zero then operation is not yet complete, goto 1
   Step 5: Else (DQ5 != 0), read DQ6 again
   Step 6: If DQ6 did not toggle between the last two reads then return
           FLASH_SUCCESS
   Step 7: Else return FLASH_TOGGLE_FAIL
*****************************************************************************/
static int FlashDataToggle( void )
{
   unsigned char uc1, uc2; /* hold values read from any address offset within
                              the Flash Memory */
```

```
   while( 1 )  /* TimeOut!: If, for some reason, the hardware fails then this
                  loop may not exit. Use a timer function to implement a timeout
                  from the loop. */
   {
      /* Step 1: Read DQ6 (into a byte) */
      uc1 = FlashRead( ANY_ADDR );   /* Read DQ6 from the Flash (any address) */

      /* Step 2: Read DQ5 and DQ6 (into another byte) */
      uc2 = FlashRead( ANY_ADDR );   /* Read DQ5 and DQ6 from the Flash (any
                                        address) */

      /* Step 3: If DQ6 did not toggle between the two reads then return
                 FLASH_SUCCESS */
      if( (uc1&0x40) == (uc2&0x40) ) /* DQ6 == NO Toggle  */
         return FLASH_SUCCESS;

      /* Step 4: Else if DQ5 is zero then operation is not yet complete */
      if( (uc2&0x20) == 0x00 )
         continue;

      /* Step 5: Else (DQ5 == 1), read DQ6 again */
      uc1 = FlashRead( ANY_ADDR );   /* Read DQ6 from the Flash (any address) */

      /* Step 6: If DQ6 did not toggle between the last two reads then
                 return FLASH_SUCCESS */
      if( (uc2&0x40) == (uc1&0x40) ) /* DQ6 == NO Toggle  */
         return FLASH_SUCCESS;

      /* Step 7: Else return FLASH_TOGGLE_FAIL */
      else                           /* DQ6 == Toggle here means fail */
         return FLASH_TOGGLE_FAIL;
   }  /* end of while loop */
}


#ifndef ILLUSTRATION_ONLY
/*****************************************************************************
Function:     static int FlashDataPoll( unsigned long ulOff,
   unsigned char ucVal )
Arguments:    ulOff should hold a valid offset to be polled. For programming
   this will be the offset of the byte being programmed. For erasing this can
   be any offset in the block(s) being erased.
   ucVal should hold the value being programmed. A value of FFh should be used
   when erasing.
Return Value: The function returns FLASH_SUCCESS if the Program/Erase Controller
   is successful or FLASH_POLL_FAIL if there is a problem.
Description: The function is used to monitor the Program/Erase Controller during
   erase or program operations. It returns when the Program/Erase Controller has
   completed. In the M29F016B Data Sheet the Data Polling Flow Chart shows the
   operation of the function.

Note: this library does not use the Data Polling Flow Chart to assess the
   correct operation of Program/Erase Controller, but uses the Data Toggle Flow
   Chart instead. The FlashDataPoll() function is only provided here as an
   illustration of the Data Polling Flow Chart in the Data Sheet.
   The code uses the function FlashDataToggle() instead.

Pseudo Code:
```

```
   Step 1: Read DQ5 and DQ7 (into a byte)
   Step 2: If DQ7 is the same as ucVal(bit 7) then return FLASH_SUCCESS
   Step 3: Else if DQ5 is zero then operation is not yet complete, goto 1
   Step 4: Else (DQ5 != 0), Read DQ7
   Step 5: If DQ7 is now the same as ucVal(bit 7) then return FLASH_SUCCESS
   Step 6: Else return FLASH_POLL_FAIL
*******************************************************************************/
static int FlashDataPoll( unsigned long ulOff, unsigned char ucVal )
{
   unsigned char uc;                /* holds value read from valid address */

   while( 1 )  /* TimeOut!: If, for some reason, the hardware fails then this
                  loop may not exit. Use a timer function to implement a timeout
                  from the loop. */
   {
      /* Step 1: Read DQ5 and DQ7 (into a byte) */
      uc = FlashRead( ulOff );             /* Read DQ5, DQ7 at valid addr */

      /* Step 2: If DQ7 is the same as ucVal(bit 7) then return FLASH_SUCCESS */
      if( (uc&0x80) == (ucVal&0x80) )      /* DQ7 == DATA  */
         return FLASH_SUCCESS;

      /* Step 3: Else if DQ5 is zero then operation is not yet complete */
      if( (uc&0x20) == 0x00 )
         continue;

      /* Step 4: Else (DQ5 == 1) */
      uc = FlashRead( ulOff );             /* Read DQ7 at valid addr */

      /* Step 5: If DQ7 is now the same as ucVal(bit 7) then
                 return FLASH_SUCCESS */
      if( (uc&0x80) == (ucVal&0x80) )      /* DQ7 == DATA  */
         return FLASH_SUCCESS;

      /* Step 6: Else return FLASH_POLL_FAIL */
      else                                 /* DQ7 != DATA here means fail */
         return FLASH_POLL_FAIL;
   }  /* end of while loop */
}
#endif /* !ILLUSTRATION_ONLY */


/*******************************************************************************
Function:     int FlashBlockFailedErase( unsigned char ucBlock )
Arguments:    ucBlock specifies the block to be checked
Return Value: FLASH_SUCCESS (-1) if the block erased successfully
   FLASH_BLOCK_FAILED_ERASE (-9) if the block failed to erase

Description:  This function can only be called after an erase operation which
   has failed the FlashDataPoll() function. It must be called before the reset
   is made.
      The function reads bit 2 of the Status Register to determine if the block
   has erased successfully or not. Successfully erased blocks should have DQ2
   set to 1 following the erase. Failed blocks will have DQ2 toggle.

Pseudo Code:
   Step 1: Read DQ2 in the block twice
   Step 2: If they are both the same then return FLASH_SUCCESS
   Step 3: Else return FLASH_BLOCK_FAILED_ERASE
```

```
*******************************************************************************/
static int FlashBlockFailedErase( unsigned char ucBlock )
{
   unsigned char ucFirstRead, ucSecondRead; /* Two variables used for clarity,
                  Optimiser will probably not use any */

   /* Step 1: Read block twice */
   ucFirstRead  = FlashRead( BlockOffset[ucBlock] ) & 0x04;
   ucSecondRead = FlashRead( BlockOffset[ucBlock] ) & 0x04;

   /* Step 2: If they are the same return FLASH_SUCCESS */
   if( ucFirstRead == ucSecondRead )
      return FLASH_SUCCESS;

   /* Step 3: Else return FLASH_BLOCK_FAILED_ERASE */
   return FLASH_BLOCK_FAILED_ERASE;
}


/*******************************************************************************
Function:     char *FlashErrorStr( int iErrNum );
Arguments:    iErrNum is the error number returned from another Flash Routine
Return Value: A pointer to a string with the error message
Description:  This function is used to generate a text string describing the
   error from the flash. Call with the return value from another flash routine.

Pseudo Code:
   Step 1: Check the error message range.
   Step 2: Return the correct string.
*******************************************************************************/
char *FlashErrorStr( int iErrNum )
{
   static char *str[] = { “Flash Success”,
                          “Flash Poll Failure”,
                          “Flash Too Many Blocks”,
                          “MPU is too slow to erase all the blocks”,
                          “Flash Block selected is invalid”,
                          “Flash Program Failure”,
                          “Flash Address Offset Out Of Range”,
                          “Flash is of Wrong Type”,
                          “Flash Block Failed Erase”,
                          “Flash is Unprotected”,
                          “Flash is Protected”,
                          “Flash function not supported”,
                          “Flash Vpp Invalid”,
                          “Flash Erase Fail”,
                          “Flash Toggle Flow Chart Failure”};

   /* Step 1: Check the error message range */
   iErrNum = -iErrNum - 1;   /* All errors are negative: make +ve & adjust */

   if( iErrNum < 0 || iErrNum >= sizeof(str)/sizeof(str[0])) /* Check range */
      return “Unknown Error\n”;

   /* Step 2: Return the correct string */
   else
      return str[iErrNum];
}
```

```
/*******************************************************************************
List of Errors and Return values, Explanations and Help.
*******************************************************************************

Return Name:  Flash_SUCCESS
Return Value: -1
Description:  This value indicates that the flash command has executed
   correctly.
*******************************************************************************

Error Name:   Flash_POLL_FAIL
Notes:        The Data Polling Flow Chart, which applies to M29 series Flash
   only, is not used in this library. The function FlashDataPoll() is only
   provided as an illustration of the Data Polling Flow Chart. This error
   condition should not occur when using this library.
Return Value: -2
Description:  The Program/Erase Controller algorithm has not managed to complete
   the command operation successfully. This may be because the device is damaged
Solution:     Try the command again. If it fails a second time then it is
   likely that the device will need to be replaced.
*******************************************************************************

Error Name:   Flash_TOO_MANY_BLOCKS
Notes:        Applies to M29 series Flash only.
Return Value: -3
Description:  The user has chosen to erase more blocks than the device has.
   This may be because the array of blocks to erase contains the same block
   more than once.
Solutions:    Check that the program is trying to erase valid blocks. The device
   will only have NUM_BLOCKS blocks (defined at the top of the file). Also check
   that the same block has not been added twice or more to the array.
*******************************************************************************

Error Name:   Flash_MPU_TOO_SLOW
Notes:        Applies to M29 series Flash only.
Return Value: -4
Description:  The MPU has not managed to write all of the selected blocks to the
   device before the timeout period expired. See BLOCK ERASE COMMAND
   section of the Data Sheet for details.
Solutions:    If this occurs occasionally then it may be because an interrupt is
   occuring between writing the blocks to be erased. Search for “DSI!” in
   the code and disable interrupts during the time critical sections.
   If this error condition always occurs then it may be time for a faster
   microprocessor, a better optimising C compiler or, worse still, learn
   assembly. The immediate solution is to only erase one block at a time.
   Disable the test (by #define’ing out the code) and always call the function
   with one block at a time.
*******************************************************************************

Error Name:   Flash_BLOCK_INVALID
Return Value: -5
Description:  A request for an invalid block has been made. Valid blocks number
   from 0 to NUM_BLOCKS-1.
Solution:     Check that the block is in the valid range.
*******************************************************************************

Error Name:   Flash_PROGRAM_FAIL
Return Value: -6
```

```
Description:  The programmed value has not been programmed correctly.
Solutions:    Make sure that the block containing the value was erased before
   programming. Try erasing the block and re-programming the value. If it fails
   again then the device may need to be changed.
*******************************************************************************

Error Name:   Flash_OFFSET_OUT_OF_RANGE
Return Value: -7
Description:  The address offset given is out of the range of the device.
Solution:     Check that the address offset is in the valid range.
*******************************************************************************

Error Name:   Flash_WRONG_TYPE
Return Value: -8
Description:  The source code has been used to access the wrong type of flash.
Solutions:    Use a different flash chip with the target hardware or contact
   STMicroelectronics for a different source code library.
*******************************************************************************

Error Name:   Flash_BLOCK_FAILED_ERASE
Return Value: -9
Description:  The previous erase to this block has not managed to successfully
   erase the block.
Solution:     Sadly the flash needs replacing.
*******************************************************************************

Return Name:  Flash_UNPROTECTED
Notes:        Applies to some M29 series Flash only. This condition should not
   occur when using this library.
Return Value: -10
Description:  The user has requested to unprotect a flash that is already
   unprotected or the user has requested to re-protect a flash that has no
   protected blocks. This is just a warning to the user that their operation
   did not make any changes and was not necessary.
*******************************************************************************

Return Name:  Flash_PROTECTED
Notes:        This condition should not occur when using this library.
Return Value: -11
Description:  The user has requested to protect a flash that is already
   protected. This is just a warning to the user that their operation did
   not make any changes and was not necessary.
*******************************************************************************

Return Name:  Flash_FUNCTION_NOT_SUPPORTED
Notes:        This condition should not occur when using this library.
Return Value: -12
Description:  The user has attempted to make use of functionality not
   available on this flash device (and thus not provided by the software
   drivers). This is simply a warning to the user.
*******************************************************************************

Error Name:   Flash_VPP_INVALID
Notes:        Applies to M28 series Flash only. This error condition should not
   occur when using this library.
Return Value: -13
Description:  A Program or a Block Erase has been attempted with the Vpp supply
   voltage outside the allowed ranges. This command had no effect since an
   invalid Vpp has the effect of protecting the whole of the flash device.
```

```
Solution:     The (hardware) configuration of Vpp will need to be modified to
   make programming or erasing the device possible.
*******************************************************************************

Error Name:   Flash_ERASE_FAIL
Return Value: -14
Description:  This indicates that the previous erasure of one block, many blocks
   or of the whole device has failed.
Solution:     Investigate this failure further by attempting to erase each block
   individually. If erasing a single block still causes failure, then the Flash
   sadly needs replacing.
*******************************************************************************

Error Name:   Flash_TOGGLE_FAIL
Return Value: -15
Notes:        This applies to M29 series Flash only.
Description:  The Program/Erase Controller algorithm has not managed to complete
   the command operation successfully. This may be because the device is damaged.
Solution:     Try the command again. If it fails a second time then it is
   likely that the device will need to be replaced.
*******************************************************************************/
```

If you have any questions or suggestion concerning the matters raised in this document please send them to the following electronic mail address:

ask.memory@st.com (for general enquiries)

Please remember to include your name, company, location, telephone number and fax number.

littlediode
COMPONENTS FOR ALL

Littlediode supplies new, hard to find or obsolete electronic components and semiconductors all over the world.

With over two million different components listed you are sure to find the part you need.

Feel free to visit us today at our online store:

LittleDiode.com

Looking forward to providing you with the best possible service.