AN1201
APPLICATION NOTE

Software Drivers for the
M59DR008 and M59DR032 Flash Memories

CONTENTS

INTRODUCTION

This application note provides library source code in C for the M59DR008 and the M59DR032 Flash Memories. Four parts in all are covered, the M59DR008E, the M59DR008F, the M59DR032A and the M59DR032B; they will all be referred to generically as the M59DRxxx.

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 c1201_16.c and c1201_16.h files contain libraries for accessing the M59DRxxx Flash Memory.

Also included in this application note is an overview of the programming model for the M59DRxxx. 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 M59DR008 or M59DR032 datasheet. It refers to the datasheet 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 M59DRxxx PROGRAMMING MODEL

The M59DR008 is an 8 Mbit (512Kb x16) Flash Memory; the M59DR032 is 32 Mbit (2Mb x 16). They can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is broken down into different blocks of varying sizes. The Main Blocks are 32Kb x16 in size, whereas the Parameter Blocks are 4Kb x16 in size. Each block can be erased individually. The blocks are grouped into two banks; each bank is independant from the other so that one can be programmed while the other is being read. All the blocks in one bank can be erased at the same time using one command.

The M59DRxxx is a smart voltage device. It differs from first generation devices which require a 12V supply to program or erase. The M59DRxxx 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. The V_{PP} pin of the M59DRxxx can be optionally supplied with 12V; this allows the double-word program command to be used and halves the programming time of 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 words to 00h before erasing to FFh using special programming sequences. The Program/Erase Controller in the M59DRxxx 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.

Bus Operations and Commands

Most of the functionality of the M59DRxxx 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 M59DRxxx. The various commands recognized by the M59DRxxx 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 M59DRxxx 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 a bank (Bank 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 at a time. Programming larger amounts of data must be done one word 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. The M59DRxxx supports double word programming when V_{PP} is at 12V. Using this feature two words can be programmed at the same time, halving the programming time. For Flash Programmers with long bus cycles the Unlock Bypass command can improve programming times. After issuing the Unlock Bypass command, Program commands only require 2 write operations and Double-Word Program commands only require 3 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 M59DRxxx 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 1s for the M59DRxxx. The Block Erase operation is resumed when the device receives the Erase Resume command.

The Status Register

While the M59DRxxx 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 M59DRxxx datasheet. 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 M59DRxxx datasheet describes the sequences of 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 int*)(0x0555) = 0x00AA;
*(unsigned int*)(0x02AA) = 0x0055;
*(unsigned int*)(0x0555) = 0x00A0;
*(unsigned int*)(0x03E2) = 0x9465;
```

This example assumes that address 0000h of the M59DRxxx 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 M59DRXXX

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 (M59DRxxx). 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 M59DRxxx. 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 M59DRxxx 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. If multiple blocks are to be erased, the function checks which bank the first block to erase is a member of and that none of the blocks specified are outside of that bank. It also checks for protected blocks and hence does not erase any blocks if they are protected or across banks.

`FlashBankErase()` is used to erase one bank in the device at a time. Before erasing the bank, this function checks that none of the blocks within that bank are protected.

`FlashBlockUnprotect()` is used to unprotect one block in the device. It checks that the block to unprotect is valid and un-protects the block. The block is then tested to ensure that the command was successful. If this function is called and the block is already un-protected the user is warned that the function had no effect.

`FlashBlockProtect()` is used to protect one block in the device. It checks that the block to protect is valid and protects the block. The block is then tested to ensure that the command was successful. If this function is called and the block is already protected the user is warned that the function had no effect.

`FlashBlockLock()` is used to lock one block in the device. Once a block is locked its protection status cannot be changed until a Hardware Reset or Power-up.

FlashProgram() is used to program data arrays into the Flash. Only previously erased words can be programmed reliably. The function will not program any data if any of the words 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.

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 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 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. Define **USE_M59DR008E**, **USE_M59DR008F**, **USE_M59DR032A** or **USE_M59DR032B** depending on which part is fitted is fitted. The top of the source file provided defines **USE_M59DR008E** 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 simply reading from the M59DRxxx. 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 M59DRxxx'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 M59DRxxx single voltage Flash Memories are ideal products for embedded and other computer systems, able to be easily interfaced to microprocessors and driven with simple software drivers written in the C language.

REVISION HISTORY

Date	Version	Revision Details
March 2001	-01	First Issue

```
/**** c1201_16.h Header File for c1201_16.c ************************************

   Filename:    c1201_16.h
   Description: Header file for c1201_16.c. Consult the C file for details

   Copyright (c) 2000 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
*******************************************************************************/

/* For use with FlashBankErase(command) */
#define BANK_A                    (0)
#define BANK_B                    (1)

/* For use with FlashAutoSelect(command) */
#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_LOCKED         (0x02) /* bit mask */
#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_UNPROTECTED          (-10)
#define FLASH_PROTECTED            (-11)
#define FLASH_ERASE_FAIL           (-14)
#define FLASH_TOGGLE_FAIL          (-15)
#define FLASH_UNPROTECT_FAIL       (-16)
#define FLASH_BANK_INVALID         (-17)
#define FLASH_PROTECT_FAIL         (-18)
#define FLASH_LOCK_FAIL            (-19)
#define FLASH_LOCKED               (-20)

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

```
Prototypes
*******************************************************************************/

extern unsigned int FlashRead( unsigned long ulOff );
extern void FlashReadReset( void );
extern int FlashAutoSelect( int iFunc );
extern int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[]);
extern int FlashBankErase( unsigned char Bank );
extern int FlashBlockUnprotect( unsigned char ucBlock );
extern int FlashBlockProtect( unsigned char ucBlock );
extern int FlashBlockLock( unsigned char ucBlock );
extern int FlashProgram( unsigned long ulOff, size_t NumWords, void *Array );
extern char *FlashErrorStr( int iErrNum );
```

```
/**** c1201_16.c ***************************************************************

   Filename:     c1201_16.c
   Description: Library routines for the M59DR008 & M59DR032 Flash parts

   Revision:     1.01
   Date:         05/02/2001
   Author:       B. Watts, Oxford Technical Solutions (www.ots.ndirect.co.uk)

   Copyright (c) 2000 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   20/09/2000  Fully tested using M59DR008E/F on target platform.
   1.01   05/02/2001  Added Flash Block Lock and retested.

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

   This source file provides library C code for using the M59DR008 and M59DR032
   devices. The following devices are supported in the code:
      M59DR008E
      M59DR008F
      M59DR032A
      M59DR032B

   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
      FlashBankErase()      to erase a whole bank
      FlashBlockUnprotect() to unprotect one block
      FlashBlockProtect()   to protect one block
      FlashProgram()        to program a word 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 word to the flash
      FlashRead()  for reading a word 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 “c1201_16.h”   /* Header file with global prototypes */

#define USE_M59DR008E

/*******************************************************************************
Constants
*******************************************************************************/
#define COUNTS_PER_MICROSECOND (200)
#define MANUFACTURER_ST (0x0020) /* Manufacturer code */
#define BASE_ADDR ((volatile unsigned int*)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 */

#ifdef USE_M59DR008E
#define EXPECTED_DEVICE (0x00A2) /* Device code for the M59DR008E */
#define FLASH_SIZE (0x80000L) /* 512K */
#endif

#ifdef USE_M59DR008F
#define EXPECTED_DEVICE (0x00A3) /* Device code for the M59DR008F */
#define FLASH_SIZE (0x80000L) /* 512K */
#endif

#ifdef USE_M59DR032A
#define EXPECTED_DEVICE (0x00A0) /* Device code for the M59DR032A */
#define FLASH_SIZE (0x200000L) /* 2048K */
#endif

#ifdef USE_M59DR032B
#define EXPECTED_DEVICE (0x00A1) /* Device code for the M59DR032B */
#define FLASH_SIZE (0x200000L) /* 2048K */
#endif

#ifdef USE_M59DR008E
```

```
/* Block organisation for Top Boot Block device */
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0  */
   0x08000L,  /* Start offset of block 1  */
   0x10000L,  /* Start offset of block 2  */
   0x18000L,  /* Start offset of block 3  */
   0x20000L,  /* Start offset of block 4  */
   0x28000L,  /* Start offset of block 5  */
   0x30000L,  /* Start offset of block 6  */
   0x38000L,  /* Start offset of block 7  */
   0x40000L,  /* Start offset of block 8  */
   0x48000L,  /* Start offset of block 9  */
   0x50000L,  /* Start offset of block 10 */
   0x58000L,  /* Start offset of block 11 */
   0x60000L,  /* Start offset of block 12 */
   0x68000L,  /* Start offset of block 13 */
   0x70000L,  /* Start offset of block 14 */
   0x78000L,  /* Start offset of block 15 */
   0x79000L,  /* Start offset of block 16 */
   0x7A000L,  /* Start offset of block 17 */
   0x7B000L,  /* Start offset of block 18 */
   0x7C000L,  /* Start offset of block 19 */
   0x7D000L,  /* Start offset of block 20 */
   0x7E000L,  /* Start offset of block 21 */
   0x7F000L   /* Start offset of block 22 */
};

/* Bank organisation for Top Boot Block device */
static const char BankFirstBlock[] =
{
   8,  /* First Block of BANK A */
   0   /* First Block of BANK B */
};

static const char BankLastBlock[] =
{
   22, /* Last Block of BANK A */
   7   /* Last Block of BANK B */

};
#endif /* USE_M59DR008E */

#ifdef USE_M59DR008F
/* Block organisation for Bottom Boot Block device */
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0  */
   0x01000L,  /* Start offset of block 1  */
   0x02000L,  /* Start offset of block 2  */
   0x03000L,  /* Start offset of block 3  */
   0x04000L,  /* Start offset of block 4  */
   0x05000L,  /* Start offset of block 5  */
   0x06000L,  /* Start offset of block 6  */
   0x07000L,  /* Start offset of block 7  */
   0x08000L,  /* Start offset of block 8  */
   0x10000L,  /* Start offset of block 9  */
   0x18000L,  /* Start offset of block 10 */
   0x20000L,  /* Start offset of block 11 */
   0x28000L,  /* Start offset of block 12 */
   0x30000L,  /* Start offset of block 13 */
   0x38000L,  /* Start offset of block 14 */
   0x40000L,  /* Start offset of block 15 */
   0x48000L,  /* Start offset of block 16 */
   0x50000L,  /* Start offset of block 17 */
   0x58000L,  /* Start offset of block 18 */
   0x60000L,  /* Start offset of block 19 */
   0x68000L,  /* Start offset of block 20 */
   0x70000L,  /* Start offset of block 21 */
   0x78000L   /* Start offset of block 22 */
};

/* Bank organisation for Bottom Boot Block device */
static const char BankFirstBlock[] =
{
   0,  /* First Block of BANK A */
   15  /* First Block of BANK B */
};

static const char BankLastBlock[] =
{
   14, /* Last Block of BANK A */
   22  /* Last Block of BANK B */

};
#endif /* USE_M59DR008F */

#ifdef USE_M59DR032A
/* Block organisation for Top Boot Block device */
static const unsigned long BlockOffset[] =
{
   0x000000L,  /* Start offset of block 0  */
   0x008000L,  /* Start offset of block 1  */
   0x010000L,  /* Start offset of block 2  */
   0x018000L,  /* Start offset of block 3  */
   0x020000L,  /* Start offset of block 4  */
   0x028000L,  /* Start offset of block 5  */
   0x030000L,  /* Start offset of block 6  */
   0x038000L,  /* Start offset of block 7  */
   0x040000L,  /* Start offset of block 8  */
   0x048000L,  /* Start offset of block 9  */
   0x050000L,  /* Start offset of block 10 */
   0x058000L,  /* Start offset of block 11 */
   0x060000L,  /* Start offset of block 12 */
   0x068000L,  /* Start offset of block 13 */
   0x070000L,  /* Start offset of block 14 */
   0x078000L,  /* Start offset of block 15 */
   0x080000L,  /* Start offset of block 16 */
   0x088000L,  /* Start offset of block 17 */
   0x090000L,  /* Start offset of block 18 */
   0x098000L,  /* Start offset of block 19 */
   0x0A0000L,  /* Start offset of block 20 */
   0x0A8000L,  /* Start offset of block 21 */
   0x0B0000L,  /* Start offset of block 22 */
   0x0B8000L,  /* Start offset of block 23 */
   0x0C0000L,  /* Start offset of block 24 */
   0x0C8000L,  /* Start offset of block 25 */
   0x0D0000L,  /* Start offset of block 26 */
   0x0D8000L,  /* Start offset of block 27 */
   0x0E0000L,  /* Start offset of block 28 */
   0x0E8000L,  /* Start offset of block 29 */
   0x0F0000L,  /* Start offset of block 30 */
   0x0F8000L,  /* Start offset of block 31 */
   0x100000L,  /* Start offset of block 32 */
   0x108000L,  /* Start offset of block 33 */
   0x110000L,  /* Start offset of block 34 */
   0x118000L,  /* Start offset of block 35 */
   0x120000L,  /* Start offset of block 36 */
   0x128000L,  /* Start offset of block 37 */
   0x130000L,  /* Start offset of block 38 */
   0x138000L,  /* Start offset of block 39 */
   0x140000L,  /* Start offset of block 40 */
   0x148000L,  /* Start offset of block 41 */
   0x150000L,  /* Start offset of block 42 */
   0x158000L,  /* Start offset of block 43 */
   0x160000L,  /* Start offset of block 44 */
   0x168000L,  /* Start offset of block 45 */
   0x170000L,  /* Start offset of block 46 */
   0x178000L,  /* Start offset of block 47 */
   0x180000L,  /* Start offset of block 48 */
   0x188000L,  /* Start offset of block 49 */
   0x190000L,  /* Start offset of block 50 */
   0x198000L,  /* Start offset of block 51 */
   0x1A0000L,  /* Start offset of block 52 */
   0x1A8000L,  /* Start offset of block 53 */
   0x1B0000L,  /* Start offset of block 54 */
   0x1B8000L,  /* Start offset of block 55 */
   0x1C0000L,  /* Start offset of block 56 */
   0x1C8000L,  /* Start offset of block 57 */
   0x1D0000L,  /* Start offset of block 58 */
   0x1D8000L,  /* Start offset of block 59 */
   0x1E0000L,  /* Start offset of block 60 */
   0x1E8000L,  /* Start offset of block 61 */
   0x1F0000L,  /* Start offset of block 62 */
   0x1F8000L,  /* Start offset of block 63 */
   0x1F9000L,  /* Start offset of block 64 */
   0x1FA000L,  /* Start offset of block 65 */
   0x1FB000L,  /* Start offset of block 66 */
   0x1FC000L,  /* Start offset of block 67 */
   0x1FD000L,  /* Start offset of block 68 */
   0x1FE000L,  /* Start offset of block 69 */
   0x1FF000L   /* Start offset of block 70 */
};

/* Bank organisation for Top Boot Block device */
static const char BankFirstBlock[] =
{
   56, /* First Block of BANK A */
   0   /* First Block of BANK B */
};

static const char BankLastBlock[] =
{
   70, /* Last Block of BANK A */
   55  /* Last Block of BANK B */

};
```

```
#endif /* USE_M59DR032A */

#ifdef USE_M59DR032B
/* Block organisation for Bottom Boot Block device */
static const unsigned long BlockOffset[] =
{
   0x000000L,  /* Start offset of block 0  */
   0x001000L,  /* Start offset of block 1  */
   0x002000L,  /* Start offset of block 2  */
   0x003000L,  /* Start offset of block 3  */
   0x004000L,  /* Start offset of block 4  */
   0x005000L,  /* Start offset of block 5  */
   0x006000L,  /* Start offset of block 6  */
   0x007000L,  /* Start offset of block 7  */
   0x008000L,  /* Start offset of block 8  */
   0x010000L,  /* Start offset of block 9  */
   0x018000L,  /* Start offset of block 10 */
   0x020000L,  /* Start offset of block 11 */
   0x028000L,  /* Start offset of block 12 */
   0x030000L,  /* Start offset of block 13 */
   0x038000L,  /* Start offset of block 14 */
   0x040000L,  /* Start offset of block 15 */
   0x048000L,  /* Start offset of block 16 */
   0x050000L,  /* Start offset of block 17 */
   0x058000L,  /* Start offset of block 18 */
   0x060000L,  /* Start offset of block 19 */
   0x068000L,  /* Start offset of block 20 */
   0x070000L,  /* Start offset of block 21 */
   0x078000L,  /* Start offset of block 22 */
   0x080000L,  /* Start offset of block 23 */
   0x088000L,  /* Start offset of block 24 */
   0x090000L,  /* Start offset of block 25 */
   0x098000L,  /* Start offset of block 26 */
   0x0A0000L,  /* Start offset of block 27 */
   0x0A8000L,  /* Start offset of block 28 */
   0x0B0000L   /* Start offset of block 29 */
   0x0B8000L,  /* Start offset of block 30 */
   0x0C0000L,  /* Start offset of block 31 */
   0x0C8000L,  /* Start offset of block 32 */
   0x0D0000L,  /* Start offset of block 33 */
   0x0D8000L,  /* Start offset of block 34 */
   0x0E0000L,  /* Start offset of block 35 */
   0x0E8000L,  /* Start offset of block 36 */
   0x0F0000L,  /* Start offset of block 37 */
   0x0F8000L,  /* Start offset of block 38 */
   0x100000L,  /* Start offset of block 39 */
   0x108000L,  /* Start offset of block 40 */
   0x110000L,  /* Start offset of block 41 */
   0x118000L,  /* Start offset of block 42 */
   0x120000L,  /* Start offset of block 43 */
   0x128000L,  /* Start offset of block 44 */
   0x130000L,  /* Start offset of block 45 */
   0x138000L,  /* Start offset of block 46 */
   0x140000L,  /* Start offset of block 47 */
   0x148000L,  /* Start offset of block 48 */
   0x150000L,  /* Start offset of block 49 */
   0x158000L,  /* Start offset of block 50 */
   0x160000L,  /* Start offset of block 51 */
   0x168000L,  /* Start offset of block 52 */
   0x170000L,  /* Start offset of block 53 */
   0x178000L,  /* Start offset of block 54 */
   0x180000L,  /* Start offset of block 55 */
   0x188000L,  /* Start offset of block 56 */
   0x190000L,  /* Start offset of block 57 */
   0x198000L,  /* Start offset of block 58 */
   0x1A0000L,  /* Start offset of block 59 */
   0x1A8000L,  /* Start offset of block 60 */
   0x1B0000L,  /* Start offset of block 61 */
   0x1B8000L,  /* Start offset of block 62 */
   0x1C0000L,  /* Start offset of block 63 */
   0x1C8000L,  /* Start offset of block 64 */
   0x1D0000L,  /* Start offset of block 65 */
   0x1D8000L,  /* Start offset of block 66 */
   0x1E0000L,  /* Start offset of block 67 */
   0x1E8000L,  /* Start offset of block 68 */
   0x1F0000L,  /* Start offset of block 69 */
   0x1F8000L   /* Start offset of block 70 */
};

/* Bank organisation for Bottom Boot Block device */
static const char BankFirstBlock[] =
{
   0,   /* First Block of BANK A */
   15   /* First Block of BANK B */
};

static const char BankLastBlock[] =
{
   14,  /* Last Block of BANK A */
   70   /* Last Block of BANK B */

};
#endif /* USE_M59DR032B */

#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))
#define NUM_BANKS  (sizeof(BankFirstBlock)/sizeof(BankFirstBlock[0]))

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

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

/******************************************************************************
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 int uVal );
#endif /* !ILLUSTRATION_ONLY */

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

```
Function:    unsigned int FlashWrite( unsigned long ulOff, unsigned int uVal)
Arguments:   ulOff is the word offset in the flash to write to
   uVal is the value to be written
Returns:     uVal
Description: This function is used to write a word 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, uVal ) ( BASE_ADDR[ulOff] = (unsigned int) uVal )

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

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

/*******************************************************************************
Function:    unsigned int FlashRead( unsigned long ulOff )
Arguments:   ulOff is the word offset into the flash to read from
Returns:     The unsigned int at the word offset
Description: This function is used to read a word 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 word offset ulOff
*******************************************************************************/
unsigned int FlashRead( unsigned long ulOff )
{
   /* Step 1 Return the value at word 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, 0x00AA );  /* 1st Cycle */
   FlashWrite( 0x02AAL, 0x0055 );  /* 2nd Cycle */
   FlashWrite( 0x0555L, 0x00F0 );  /* 3rd Cycle */

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

/*******************************************************************************
Function:      int FlashAutoSelect( int iFunc )
Arguments:     iFunc should be set to either 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 0020h.

   When iFunc is FLASH_READ_DEVICE_CODE (-1) the function returns the Device
   Code.  The device codes for the parts are:
      M59DR008E   00A2h
      M59DR008F   00A3h
      M59DR032A   00A0h
      M59DR032B   00A1h

   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, 0x00AA );  /* 1st Cycle */
   FlashWrite( 0x02AAL, 0x0055 );  /* 2nd Cycle */
   FlashWrite( 0x0555L, 0x0090 );  /* 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,0x00F0);

   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 block found which is protected, invalid, or outside the
   particular bank.

   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 as long as they are within the same bank.
   The function does not return until the blocks are erased.
   If any blocks are protected,invalid or in different banks none of the
   blocks will be 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:  Discover which bank the first block to erase is in then ensure that
            other blocks listed in the array are also within that bank.
   Step 4:  Write Block Erase command.
   Step 5:  Check for time-out blocks.
   Step 6:  Wait for the timer bit to be set.
   Step 7:  Follow Data Toggle Flow Chart until Program/Erase Controller has
            completed.
   Step 8:  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 */
   unsigned char ucCurBank;/* Range Variable to track current bank */
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
   unsigned int uFirstRead, uSecondRead; /* 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: Find which bank the first block in the array is member of */
   for( ucCurBank = 0; ucCurBank < NUM_BANKS; ucCurBank++ )
      if ( (ucBlock[0] >= BankFirstBlock[ucCurBank]) &&
          (ucBlock[0] <= BankLastBlock[ucCurBank]) )
        break; /* ucCurBank is the current bank of block 0 */

   /* Check that all the other blocks are within the bank ucCurBank */
   for( ucCurBlock = 1; ucCurBlock < ucNumBlocks; ucCurBlock++ )
      if ( (ucBlock[ucCurBlock] < BankFirstBlock[ucCurBank]) ||
           (ucBlock[ucCurBlock] > BankLastBlock[ucCurBank]) )
```

```
     return ucBlock[ucCurBlock];  /* Return invalid blocks that are across
                                     banks */

   /* Step 4: Write Block Erase command */
   FlashWrite( 0x0555L, 0x00AA );
   FlashWrite( 0x02AAL, 0x0055 );
   FlashWrite( 0x0555L, 0x0080 );
   FlashWrite( 0x0555L, 0x00AA );
   FlashWrite( 0x02AAL, 0x0055 );
   /* DSI!: Time critical section. Additional blocks must be added every 100us*/
   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      FlashWrite( BlockOffset[ucBlock[ucCurBlock]], 0x0030 );

      /* Check for Erase Timeout Period (is bit DQ3 set?) */
      if( (FlashRead( BlockOffset[ucBlock[0]] ) & 0x0008) == 0x0008 )
         break; /* Cannot set any more blocks due to timeout */
   }
   /* ENI! */

   /* Step 5: 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 )
   {
      uFirstRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x0004;
      uSecondRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x0004;
      if( uFirstRead != uSecondRead )
      {
         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 6: 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]] ) & 0x0008 ) == 0x0008 )
         break; /* Break when device starts the erase cycle */
   }

   /* Step 7: Follow Data Toggle Flow Chart until Program/Erase Controller
              completes */
   if( FlashDataToggle( BlockOffset[ucBlock[0]] ) != FLASH_SUCCESS )
   {
      iRetVal = FLASH_ERASE_FAIL;
      /* Step 8: Return to Read mode (if an error occurred) */
      FlashReadReset();
   }

   return iRetVal;
}

/*******************************************************************************
Function:     int FlashBankErase( unsigned char ucBank )
Arguments:    ucBank is 0 for Bank A, 1 For Bank B
Return Value: The function returns the following conditions:
   FLASH_SUCCESS         (-1)
   FLASH_WRONG_TYPE      (-8)
   FLASH_ERASE_FAIL      (-14)
   FLASH_BANK_INVALID    (-17)
   The first block in the bank found to be protected will cause the funtion
   to fail returning the block number that failed.
Description:  The function can be used to erase a whole bank of the flash chip.
   Each Bank is erased in turn. The function only returns when all of the Bank
   has been erased or if a error has been detected.

Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: Check that the bank value ucBank in not invalid
   Step 3: Check for protect blocks in the bank
   Step 4: Write bank erase sequence
   Step 5: Use Data Toggle to check for completetion
   Step 6: Return Fail if Data Toggle fails
   Step 7: Read DQ5 to see if P/EC failed
   Step 8: Return success
*******************************************************************************/
int FlashBankErase( unsigned char ucBank )
{
   unsigned char ucCurBlock; /* Variable for checking for protected blocks */

   /* 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 bank value is not greater than NUM_BANKS */
   if( ucBank > NUM_BANKS )
      return FLASH_BANK_INVALID;

   /* Step 3: Check for protected blocks in that bank */
   for( ucCurBlock = BankFirstBlock[ucBank];
      ucCurBlock <= BankLastBlock[ucBank]; ucCurBlock++ )
   {
      /* Use FlashAutoSelect to find protected or invalid blocks */
      if( FlashAutoSelect( ucCurBlock ) != FLASH_BLOCK_UNPROTECTED )
         return (int)ucCurBlock; /* Return protected/invalid blocks */
   }

   /* Step 4: Write command sequence to erase bank */
   FlashWrite( 0x0555, 0x00AA );
   FlashWrite( 0x02AA, 0x0055 );
   FlashWrite( 0x0555, 0x0080 );
   FlashWrite( 0x0555, 0x00AA );
   FlashWrite( 0x02AA, 0x0055 );
   FlashWrite( BankFirstBlock[ucBank], 0x0010 );

   /* Step 5: Follow Data Toggle Flow Chart until Program/Erase Controller
      completes */
   if( FlashDataToggle( BankFirstBlock[ucBank] ) != FLASH_SUCCESS )

    {
   /* Step 6: Return to Read mode (if an error occurred) */
      FlashReadReset();
      return FLASH_ERASE_FAIL;
    }

   /* Step 7: Pass */
   return FLASH_SUCCESS;
}

/*******************************************************************************
Function:     int FlashBlockUnprotect(unsigned char ucBlock)
Arguments:    ucBlock holds the block number to unprotect
Return Value: The function returns the following conditons:
   FLASH_WRONG_TYPE     (-8)
   FLASH_SUCCESS        (-1)
   FLASH_UNPROTECT_FAIL (-16)
   FLASH_UNPROTECTED    (-10)
   FLASH_BLOCK_INVALID  (-5)
Description:   This function unprotects a block selected by ucBlock but only if
   that particular block is protected and valid. The block unprotect command
   is then written and then checked to ensure that it was successful.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check to see if the block number ucBlock is valid.
   Step 3:  Check if the block really needs unprotecting as it may already be
            unprotected.
   Step 4:  Unprotect the block.
   Step 5:  Test to see if Unprotection was successful.
   Step 6:  Read reset if Data Toggle Flow Chart failed.
*******************************************************************************/
int FlashBlockUnprotect(unsigned char ucBlock)
{
   int iRetVal; /* Store result */

   /* 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;

   iRetVal = FlashAutoSelect( ucBlock );

   /* Step 2 */
   if( iRetVal == FLASH_BLOCK_INVALID ) return iRetVal;

   /* Step 3 */
   else if( iRetVal == FLASH_BLOCK_UNPROTECTED ) return FLASH_UNPROTECTED;

   else
   {
     /* Step 4 Unprotect the block */
     FlashWrite( 0x0555, 0x00AA );
     FlashWrite( 0x02AA, 0x0055 );
     FlashWrite( 0x0555, 0x0060 );
     FlashWrite( BlockOffset[ucBlock], 0x00D0 );


     /* Step 5 test to see that it worked */
     if( FlashAutoSelect( ucBlock ) == FLASH_BLOCK_UNPROTECTED )
        return FLASH_SUCCESS;
     else
     {
        /* Step 6 Read reset if failure */
        FlashReadReset();
        return FLASH_UNPROTECT_FAIL;
     }
   }
}

/*******************************************************************************
Function:     int FlashBlockProtect(unsigned char ucBlock)
Arguments:    ucBlock holds the block number to protect
Return Value: The function returns the following conditons:
   FLASH_SUCCESS        (-1)
   FLASH_PROTECT_FAIL   (-18)
   FLASH_PROTECTED      (-11)
   FLASH_WRONG_TYPE     (-8)
   FLASH_BLOCK_INVALID  (-5)
Description:  This function protects a block selected by ucBlock but only if
   that particular block is unprotected and valid. The block protect command
   is then written and checked to ensure that it was successful.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check to see if the block number ucBlock is valid
   Step 3:  Check if the block really needs protecting as it may already be
            protected.
   Step 4:  Protect the block.
   Step 5:  Test to see if Protection was successful.
   Step 6:  Read reset if Data Toggle Flow Chart failed.
*******************************************************************************/
int FlashBlockProtect(unsigned char ucBlock)
{
   int iRetVal; /* Store result */

   /* 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;

   iRetVal = FlashAutoSelect( ucBlock );

   /* Step 2 */
   if( iRetVal == FLASH_BLOCK_INVALID ) return iRetVal;

   /* Step 3 */
   else if( iRetVal == FLASH_BLOCK_PROTECTED ) return FLASH_PROTECTED;

   else
   {
     /* Step 4 Protect the block */
     FlashWrite( 0x0555, 0x00AA );
     FlashWrite( 0x02AA, 0x0055 );
     FlashWrite( 0x0555, 0x0060 );
     FlashWrite( BlockOffset[ucBlock], 0x0001 );

     /* Step 5 test to see that it worked */
     if ( FlashAutoSelect( ucBlock ) == FLASH_BLOCK_PROTECTED )
        return FLASH_SUCCESS;
     else
     {
        /* Step 6 Read reset if failure */
        FlashReadReset();
        return FLASH_UNPROTECT_FAIL;
     }
   }
}

/*******************************************************************************
Function:     int FlashBlockLock(unsigned char ucBlock)
Arguments:    ucBlock holds the block number to lock
Return Value: The function returns the following conditons:
   FLASH_SUCCESS        (-1)
   FLASH_LOCK_FAIL      (-19)
   FLASH_LOCKED         (-20)
   FLASH_WRONG_TYPE     (-8)
   FLASH_BLOCK_INVALID  (-5)
Description:  This function locks a block selected by ucBlock but only if
   that particular block is valid. The block lock command is then written and
   checked to ensure that it was successful.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check to see if the block number ucBlock is valid
   Step 3:  Check if the block really needs locking as it may already be
            locked.
   Step 4:  Lock the block.
   Step 5:  Test to see if Lock was successful.
   Step 6:  Read reset if Data Toggle Flow Chart failed.
*******************************************************************************/
int FlashBlockLock(unsigned char ucBlock)
{
   int iRetVal; /* Store result */

   /* 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;

   iRetVal = FlashAutoSelect( ucBlock );

   /* Step 2 */
   if( iRetVal == FLASH_BLOCK_INVALID ) return iRetVal;

   /* Step 3 */
   if( ( iRetVal & FLASH_BLOCK_LOCKED ) ) return FLASH_LOCKED;

   else
   {
     /* Step 4 Lock the block */
```

```
      FlashWrite( 0x0555, 0x00AA );
      FlashWrite( 0x02AA, 0x0055 );
      FlashWrite( 0x0555, 0x0060 );
      FlashWrite( BlockOffset[ucBlock], 0x002F );

      /* Step 5 test to see that it worked */
      if ( ( FlashAutoSelect( ucBlock ) & FLASH_BLOCK_LOCKED ) )
         return FLASH_SUCCESS;
      else
      {
         /* Step 6 Read reset if failure */
         FlashReadReset();
         return FLASH_LOCK_FAIL;
      }
   }
}

/*******************************************************************************
Function:     int FlashProgram( unsigned long ulOff, size_t NumWords,
   void *Array )
Arguments:    ulOff is the word offset into the flash to be programmed
   NumWords holds the number of words 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 word.
   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 NumWords, void *Array )
{
   unsigned int *uArrayPointer;   /* Use an unsigned int 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+NumWords-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 */
   uArrayPointer = (unsigned int *)Array;
   while( ulOff <= ulLastOff )
   {
      /* Step 5: Check for changes from ‘0’ to ‘1’ */
      if( ~FlashRead( ulOff ) & *uArrayPointer )
         /* 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 word */
      FlashWrite( 0x0555L, 0x00AA );  /* 1st cycle */
      FlashWrite( 0x02AAL, 0x0055 );  /* 2nd cycle */
      FlashWrite( 0x0555L, 0x00A0 );  /* Program command */
      FlashWrite( ulOff, *uArrayPointer );   /* 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( ulOff ) == FLASH_TOGGLE_FAIL )
      {
         /* Step 8: Return to Read mode (if an error occurred) */
         FlashReadReset();
         return FLASH_PROGRAM_FAIL;
      }

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

   return FLASH_SUCCESS;
```

```
}

/*******************************************************************************
Function:     static int FlashDataToggle( void )
Arguments:    ulOff should hold a valid offset for reading the Status Register.
   This address offset must be within the bank in which the current erase or
   program operation is taking place.
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 M59DR008 Data Sheet or the M59DR032 Data
   Sheet, the Data Toggle Flow Chart shows the operation of the function.

Pseudo Code:
   Step 1: Read DQ6 (into a word).
   Step 2: Read DQ5 and DQ6 (into another word).
   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( unsigned long ulOff )
{
   unsigned int u1, u2; /* hold values read from address offset ulOff */

   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 word) */
      u1 = FlashRead( ulOff );    /* Read DQ6 from the Flash (any address) */

      /* Step 2: Read DQ5 and DQ6 (into another word) */
      u2 = FlashRead( ulOff );    /* 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( (u1&0x0040) == (u2&0x0040) )   /* DQ6 == NO Toggle  */
         return FLASH_SUCCESS;

      /* Step 4: Else if DQ5 is zero then operation is not yet complete */
      if( (u2&0x0020) == 0x0000 )
         continue;

      /* Step 5: Else (DQ5 == 1), read DQ6 again */
      u1 = FlashRead( ulOff );    /* Read DQ6 from the Flash (any address) */

      /* Step 6: If DQ6 did not toggle between the last two reads then
                 return FLASH_SUCCESS */
      if( (u2&0x0040) == (u1&0x0040) )   /* 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 int uVal )
Arguments:    ulOff should hold a valid offset to be polled. For programming
   this will be the offset of the word being programmed. For erasing this can
   be any offset in the block(s)/bank being erased.
   uVal 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 M59DR008 Data Sheet or the M59DR032 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 word).
   Step 2: If DQ7 is the same as uVal(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 uVal(bit 7) then return FLASH_SUCCESS.
   Step 6: Else return FLASH_POLL_FAIL.
*******************************************************************************/
static int FlashDataPoll( unsigned long ulOff, unsigned int uVal )
{
   unsigned int u;                 /* 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 word) */
      u = FlashRead( ulOff );                /* Read DQ5, DQ7 at valid addr */

      /* Step 2: If DQ7 is the same as Value(bit 7) then return FLASH_SUCCESS */
      if( (u&0x0080) == (uVal&0x0080) )      /* DQ7 == DATA  */
         return FLASH_SUCCESS;

      /* Step 3: Else if DQ5 is zero then operation is not yet complete */
      if( (u&0x0020) == 0x0000 )
         continue;

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

      /* Step 5: If DQ7 is now the same as uVal(bit 7) then
                 return FLASH_SUCCESS */
      if( (u&0x0080) == (uVal&0x0080) )      /* 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:     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”,
                          “Flash Unprotect Failed”,
                          “Flash Bank Invalid”,
                          “Flash Protect Failed”,
                          “Flash Lock Failed”,
                          “Flash is Locked”
                         };

   /* 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 M59 and 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
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
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
Return Value: -10
Description:  The user has requested to unprotect a flash that is already
   unprotected. This is just a warning to the user that their operation
   did not make any changes and was not necessary.
*******************************************************************************

Return Name:  FLASH_PROTECTED
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:        This 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 a bank of the 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 and M59 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.
********************************************************************************

Error Name:   FLASH_UNPROTECT_FAIL
Return Value: -16
Notes:        This applies to M59DRxxx series Flash only.
Description:  This error return value indicates that a block unprotect command
   was unsuccessful.
Solution:     Try the command again. If it fails a second time then it is
   likely that the device is either locked or will need to be replaced.
   (Part is unlocked but protected on power-up with /WP pin at V_high).
********************************************************************************

Error Name:   FLASH_BANK_INVALID
Return Value: -17
Notes:        This applies to M59DRxxx series Flash only.
Description:  This error return value indicates that a bank does not exist.
********************************************************************************

Error Name:   FLASH_PROTECT_FAIL
Return Value: -18
Notes:        This applies to M59DRxxx series Flash only.
Description:  This error return value indicates that a block protect command
   was unsuccessful.
Solution:     Try the command again. If it fails a second time then it is
   likely that the device is either locked or will need to be replaced.
   (Part is unlocked but protected on power-up with /WP pin at V_high).
********************************************************************************

Error Name:   FLASH_LOCK_FAIL
Return Value: -19
Notes:        This applies to M59DRxxx series Flash only.
Description:  This error return value indicates that the Lock Bit for a
   particular block could not be set.
********************************************************************************

Error Name:   FLASH_LOCKED
Return Value: -20
Notes:        This applies to M59DRxxx series Flash only.
Description:  This error return value indicates that the block was already
   locked when the attempt to lock it was made.
*******************************************************************************/
```

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.