AN1203
APPLICATION NOTE

Software Drivers for the M28W800C, M28W160C and M28W320C Flash Memories

CONTENTS

- INTRODUCTION
- THE M28WxxxC PROGRAMMING MODEL
- WRITING C CODE FOR THE M28WxxxC
- C LIBRARY FUNCTIONS PROVIDED
- PORTING THE DRIVERS TO THE TARGET SYSTEM
- LIMITATIONS OF THE SOFTWARE
- CONCLUSION
- c1203_16.h LISTING
- c1203_16.c LISTING

INTRODUCTION

This application note provides library source code in C for the M28W800CB, M28W800CT, M28W160CB, M28W160CT, M28W320CB and M28W320CT Flash Memories. There are two types of each Flash Memory, a Top Boot Block and a Bottom Boot Block, both types are covered in the C code. The set of Flash Memories will be referred to as M28WxxxC throughout this document.

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 c1203_16.c and c1203_16.h files contain libraries for accessing the M28WxxxC Flash Memories.

Also included in this application note is an overview of the programming model for the M28WxxxC. 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 M28WxxxC 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 M28WxxxC provides all the hardware and software functionality available on Intel's Advanced Boot Block Flash Memories. Source code written for Intel's Advanced Boot Block Flash Memories can easily be modified to use STMicroelectronics' M28WxxxC memory instead.

The M28WxxxC Flash Memory is compatible with the JEDEC Common Flash Interface (CFI). This interface allows software to identify the physical memory layout and command set of the Flash Memory. Software that is CFI compatible and includes the correct programming algorithm will automatically be able to

identify and operate the memory. In this case the software drivers accompanying this application note will not be required. Note that the software drivers provided with this application note are not CFI drivers and that they are targeted solely at the M28WxxxC Flash Memory.

THE M28WxxxC PROGRAMMING MODEL

The M28WxxxC 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 protected and unprotected to prevent or allow Progam or Erase commands from affecting the memory, the protection state is volatile.

The M28WxxxC is a smart voltage device. It differs from first generation devices which require a 12V supply to program or erase. The M28WxxxC 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. Three power supply pins are used to give the optimal supply voltage conditions. In the simplest model all three voltage supply pins can be supplied with 2.7 to 3.6V. All operations are available in this mode. Alternatively the input/output driver section (supplied by V_{CCQ}) can be supplied down to 1.65V to interface to 1.8V Microcontrollers. The program/erase section can be supplied with 12V to speed up programming in the factory; this mode is not to be used as a permanent solution.

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 0000h before erasing to FFFFh using special programming sequences. The Program/Erase Controller in the M28WxxxC 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 M28WxxxC is available via the two standard bus operations: read and write. Read operations retrieve data or status information from the device. Write operations are interpreted by the device as commands, which modify the data stored or the behavior of the device. Only certain special sequences of write operations are recognized as commands by the M28WxxxC. The various commands recognized by the M28WxxxC are listed in the Instructions Table of the datasheet; the main commands can be grouped as follows:

1. Read
2. Read Electronic Signature
3. Erase
4. Program
5. Program/Erase Suspend
6. Common Flash Interface Query
7. Block Protect/Unprotect

The Read command returns the M28WxxxC 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 Read Electronic Signature command places the device in a 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. In the library of software drivers this mode is known as the Auto Select mode to make the M28WxxxC compatible with the M29 series Flash Memories.

The Erase command is used to set all the bits to ‘1’ in every memory location in the selected block. All data previously stored in the erased block will be lost. The erase command takes longer to execute than

the other commands, because an entire block is erased at once. Attempts to erase or program a protected block generate an error and do not modify the contents of the memory.

The Program command is used to modify the data stored at the specified address of the device. Note that programming can only change bits from '1' to '0'. Attempting to change a '0' to a '1' using the Program command will fail. 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.

Issuing the Program/Erase Suspend command during a Program or Erase operation will temporarily place the M28WxxxC in Program/Erase Suspend mode. While an Erase operation is being suspended the blocks not being erased may be read or programmed as if in the reset state of the device. While a Program operation is being suspended the rest of the device may be read. This allows the user to access information stored in the device immediately rather than waiting until the Program or Erase operation completes, typically 10µs for programming and 1s for erasing on the M28WxxxC. The Program or Erase operation is resumed when the device receives the Program/Erase Resume command.

The Common Flash Interface Query command of the device allows the user to identify the number of blocks and the addresses of the blocks in the Flash. The interface also contains information relating to the typical and maximum program and erase durations. This allows the user to implement software timeouts instead of waiting indefinitely for a defective Flash to finish programming or erasing. For further information about the CFI, please refer to the CFI specification available from the internet site (http://www.st.com) or from your STMicroelectronics distributor.

Blocks can be protected against accidental or malicious Program and Erase operations changing their contents. Block protection is volatile; after power up or a hardware reset all blocks are protected, but not locked. The boot code can lock blocks that require additional security and unlock user data blocks before the application runs, giving full flexible control over block protection.

The Status Register

While the M28WxxxC is programming or erasing, a read from the device will output the Status Register of the Program/Erase Controller. The Status Register, which can also be accessed by issuing the Read Status Register command, provides valuable information about the most recent Program or Erase command. The Status Register bits are described in the Status Register Bits Table of the M28WxxxC 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 is indicated by the Program/Erase Controller Status bit (Status Register bit DQ7) becoming '1'. Programming or erasing errors are then indicated by one or more of the various error bits (Status Register bits DQ1, DQ3, DQ4 and DQ5) being '1'. If a failure occurs the Status Register error bits will remain set until a Clear Status Register command is issued to the device. This should be done before performing any further operations or it will not be possible to determine whether the following operation resulted in an error or not.

A Detailed Example

The Instructions Table of the M28WxxxC datasheet describes the sequences of write operations that will be recognized by the Program/Erase Controller as valid commands. For example programming 9465h to the address 03E2h requires the user to write the following sequence (in C):

```
*(unsigned int*)(0x0000) = 0x0040;
*(unsigned int*)(0x03E2) = 0x9465;
```

The first of the two addresses (0000h) is arbitrary, so long as it is inside the Flash address space. This example assumes that address 0000h of the M28WxxxC is mapped to address 0000h in the microproces-

sor 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 DQ7 will be '0' while programming is on-going and will become '1' on completion. If Status Register bits DQ1, DQ3 or DQ4 are set on completion then the Program command will have failed. The Status Register bits do not always indicate an error when an attempt to change a '0' to a '1' has been made; it is recommended that the value in the memory after programming is compared to the desired value to trap this error.

WRITING C CODE FOR THE M28WxxxC

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 (M28WxxxC). 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 M28WxxxC. 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 M28WxxxC 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 Array 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 Block Protection Status of the device. The function uses the Read Electronic Signature mode of the device. The function is called **FlashAutoSelect()** to make it compatible with the M29 series Flash Memories.

FlashReadCFI() is used to read a word from the CFI area of the memory. After reading the CFI word the memory is returned to Read mode.

FlashBlockErase() is used to erase a block in the device. Protected blocks cannot be erased and the function returns an error when the user attempts to erase a protected block. Similarly, the blocks cannot be erased when V_{PP} is invalid: attempting to do so generates an error and leaves the Flash in an indeterminate state.

FlashChipErase() is used to erase the entire chip. It cannot erase any protected blocks. If some of the blocks are protected an error is returned. The unprotected blocks are still erased.

FlashProgram() is used to program data arrays into the Flash. Only previously erased words can be programmed reliably. Again, protected blocks cannot be programmed, nor can programming take place when V_{PP} is invalid.

FlashBlockProtect() is used to protect a block, preventing Program and Erase operations from changing the contents.

FlashBlockUnprotect() is used to unprotect a block, allowing Program and Erase operations to change the contents.

FlashBlockLock() is used to lock a block, see the Protection States table of the datasheet. Once locked the lock state of a block can only be removed through a Hardware Reset.

The functions provided in the software library rely on the user implementing the hardware specific bus operations. This is to be done by writing two 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.

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..D15 of the device on D16..D31 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 (particularly in the user's **FlashRead()** function).

A **long** is 32 bits (4 bytes). It is necessary to have arithmetic greater than 16 bits in order to address the entire device.

Two approaches to the addressing are available: the desired address in the Flash can be specified by a 32 bit linear pointer or a 32 bit offset into the device could be provided by the user. The **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 one of the following depending on which part is fitted. The top of the source file provided defines **USE_M28W800CT** as an example.

   USE_M28W800CB

   USE_M28W800CT

   USE_M28W160CB

   USE_M28W160CT

   USE_M28W320CB

   USE_M28W320CT

2. Write **FlashRead()** and **FlashWrite()** functions appropriate to the Target Hardware.

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 M28WxxxC. If it is erased then only FFFFh 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. Two 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 all the functions.

The programmer should also take care when a Hardware Reset is applied during Program or Erase operations. The Flash will be left in an indeterminate state and data could be lost.

LIMITATIONS OF THE SOFTWARE

The software provided does not implement a full set of the M28WxxxC's functionality. It is left to the user to implement the Program/Erase Suspend command of the device. The Double Word Program and the Protection Register Program commands are intended for use but programming equipment and have not been included in these embedded drivers. 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 M28WxxxC 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.

REVISION HISTORY

Date	Version	Revision Details
December - 2000	-01	First Issue
18-Sep-2001	-02	Changes to pages 12, 13 and 14. Top Boot Block numbers reversed to correspond to datasheet.

```
/**** c1203_16.h Header File for Flash Memory ********************************

   Filename:    c1203_16.h
   Description: Header file for c1203_16.c V1.01. 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
******************************************************************************/
#define FLASH_BLOCK_LOCKED_PROTECTED (0x03)
#define FLASH_BLOCK_LOCKED           (0x02)
#define FLASH_BLOCK_PROTECTED        (0x01)
#define FLASH_BLOCK_UNPROTECTED      (0x00)

#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_SUCCESS                 (-1)
#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_FUNCTION_NOT_SUPPORTED  (-12)
#define FLASH_VPP_INVALID             (-13)
#define FLASH_ERASE_FAIL              (-14)
#define FLASH_UNPROTECT_FAIL          (-16)
#define FLASH_PROTECT_FAIL            (-18)
#define FLASH_CFI_FAIL                (-19)
#define FLASH_LOCKED                  (-20)
#define FLASH_LOCK_FAIL               (-21)

/******************************************************************************
Function Prototypes
******************************************************************************/
extern unsigned int FlashRead( unsigned long ulOff );
extern void FlashReadReset( void );
extern int FlashAutoSelect( int iFunc );
extern int FlashReadCFI( int iCFIFunc, unsigned int *uCFIValue );
extern int FlashBlockErase( unsigned char ucBlock );
extern int FlashChipErase( int *iResults );
extern int FlashBlockProtect(unsigned char ucBlock);
extern int FlashBlockLock(unsigned char ucBlock);
```

```
extern int FlashBlockUnprotect(unsigned char ucBlock);
extern int FlashProgram( unsigned long ulOff, size_t NumWords, void *Array );
extern char *FlashErrorStr( int iErrNum );
```

```
/**** c1203_16.c  M28WxxxC Flash Memory Driver ********************************

   Filename:    c1203_16.c
   Description: Library routines for the M28WxxxC Flash Memory.

   Version:     1.00 Initial release.
   Date:        10/11/2000
   Author:      Tim Webster, 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

   0.00   18/09/2000  Initial creation

   1.00   25/09/2000  Software tested using M28W032CB.

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

   This source file provides library C code for using the M28WxxxC devices.
   The following devices are supported in the code:
      M28W800CT
      M28W800CB
      M28W160CT
      M28W160CB
      M28W320CT
      M28W320CB

   The following functions are available in this library:

      FlashReadReset()      to reset the flash for normal memory access
      FlashAutoSelect()     to get information about the device
      FlashReadCFI()        to read CFI information from the flash
      FlashBlockErase()     to erase a single block
      FlashChipErase()      to erase the whole chip
      FlashBlockProtect()   to protect a block in the flash
      FlashBlockLock()      to lock a block and prevent it being unprotected
      FlashBlockUnprotect() to unprotect a block in the flash
      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
```

```
   A list of the error conditions is 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.

   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 “c1203_16.h”   /* Header file with global prototypes */

#define USE_M28W320CB

/*******************************************************************************
Constants
*******************************************************************************/
#define MANUFACTURER_ST (0x0020)  /* ST RSIG for manufacturer is always 0x20 */
#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_M28W800CT
#define EXPECTED_DEVICE (0x88CC) /* Device code for the M28W800CT */
#define FLASH_SIZE (0x80000L) /* 512K */
#endif

#ifdef  USE_M28W800CB
#define EXPECTED_DEVICE (0x88CD) /* Device code for the M28W800CB */
#define FLASH_SIZE (0x80000L) /* 512K */
#endif

#ifdef  USE_M28W160CT
#define EXPECTED_DEVICE (0x88CE) /* Device code for the M28W160CT */
#define FLASH_SIZE (0x100000L) /* 1M */
#endif

#ifdef  USE_M28W160CB
#define EXPECTED_DEVICE (0x88CF) /* Device code for the M28W160CB */
#define FLASH_SIZE (0x100000L) /* 1M */
#endif

#ifdef  USE_M28W320CT
#define EXPECTED_DEVICE (0x88BA) /* Device code for the M28W160CT */
#define FLASH_SIZE (0x200000L) /* 2M */
#endif
```

```
#ifdef  USE_M28W320CB
#define EXPECTED_DEVICE (0x88BB) /* Device code for the M28W160CB */
#define FLASH_SIZE (0x200000L) /* 2M */
#endif

#ifdef USE_M28W800CT
/* Block organisation for Top Boot Block devices */
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 22 */
   0x08000L,  /* Start offset of block 21 */
   0x10000L,  /* Start offset of block 20 */
   0x18000L,  /* Start offset of block 19 */
   0x20000L,  /* Start offset of block 18 */
   0x28000L,  /* Start offset of block 17 */
   0x30000L,  /* Start offset of block 16 */
   0x38000L,  /* Start offset of block 15 */
   0x40000L,  /* Start offset of block 14 */
   0x48000L,  /* Start offset of block 13 */
   0x50000L,  /* Start offset of block 12 */
   0x58000L,  /* Start offset of block 11 */
   0x60000L,  /* Start offset of block 10 */
   0x68000L,  /* Start offset of block 9 */
   0x70000L,  /* Start offset of block 8 */
   0x78000L,  /* Start offset of block 7 */
   0x79000L,  /* Start offset of block 6 */
   0x7A000L,  /* Start offset of block 5 */
   0x7B000L,  /* Start offset of block 4 */
   0x7C000L,  /* Start offset of block 3 */
   0x7D000L,  /* Start offset of block 2 */
   0x7E000L,  /* Start offset of block 1 */
   0x7F000L   /* Start offset of block 0 */
};
#endif /* USE_M28W800CT */

#ifdef USE_M28W800CB
/* Block organisation for Bottom Boot Block devices */
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 */
};
#endif /* USE_M28W800CB */

#ifdef USE_M28W160CT
/* Block organisation for Top Boot Block devices */
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 38 */
   0x08000L,  /* Start offset of block 37 */
   0x10000L,  /* Start offset of block 36 */
   0x18000L,  /* Start offset of block 35 */
   0x20000L,  /* Start offset of block 34 */
   0x28000L,  /* Start offset of block 33 */
   0x30000L,  /* Start offset of block 32 */
   0x38000L,  /* Start offset of block 31 */
   0x40000L,  /* Start offset of block 30 */
   0x48000L,  /* Start offset of block 29 */
   0x50000L,  /* Start offset of block 28 */
   0x58000L,  /* Start offset of block 27 */
   0x60000L,  /* Start offset of block 26 */
   0x68000L,  /* Start offset of block 25 */
   0x70000L,  /* Start offset of block 24 */
   0x78000L,  /* Start offset of block 23 */
   0x80000L,  /* Start offset of block 22 */
   0x88000L,  /* Start offset of block 21 */
   0x90000L,  /* Start offset of block 20 */
   0x98000L,  /* Start offset of block 19 */
   0xA0000L,  /* Start offset of block 18 */
   0xA8000L,  /* Start offset of block 17 */
   0xB0000L,  /* Start offset of block 16 */
   0xB8000L,  /* Start offset of block 15 */
   0xC0000L,  /* Start offset of block 14 */
   0xC8000L,  /* Start offset of block 13 */
   0xD0000L,  /* Start offset of block 12 */
   0xD8000L,  /* Start offset of block 11 */
   0xE0000L,  /* Start offset of block 10 */
   0xE8000L,  /* Start offset of block 9 */
   0xF0000L,  /* Start offset of block 8 */
   0xF8000L,  /* Start offset of block 7 */
   0xF9000L,  /* Start offset of block 6 */
   0xFA000L,  /* Start offset of block 5 */
   0xFB000L,  /* Start offset of block 4 */
   0xFC000L,  /* Start offset of block 3 */
   0xFD000L,  /* Start offset of block 2 */
   0xFE000L,  /* Start offset of block 1 */
   0xFF000L   /* Start offset of block 0 */
};
#endif /* USE_M28W160CT */

#ifdef USE_M28W160CB
/* Block organisation for Bottom Boot Block devices */
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 */
   0x80000L,  /* Start offset of block 23 */
   0x88000L,  /* Start offset of block 24 */
   0x90000L,  /* Start offset of block 25 */
   0x98000L,  /* Start offset of block 26 */
   0xA0000L,  /* Start offset of block 27 */
   0xA8000L,  /* Start offset of block 28 */
   0xB0000L,  /* Start offset of block 29 */
   0xB8000L,  /* Start offset of block 30 */
   0xC0000L,  /* Start offset of block 31 */
   0xC8000L,  /* Start offset of block 32 */
   0xD0000L,  /* Start offset of block 33 */
   0xD8000L,  /* Start offset of block 34 */
   0xE0000L,  /* Start offset of block 35 */
   0xE8000L,  /* Start offset of block 36 */
   0xF0000L,  /* Start offset of block 37 */
   0xF8000L   /* Start offset of block 38 */
};
#endif /* USE_M28W160CB */

#ifdef USE_M28W320CT
/* Block organisation for Top Boot Block devices */
static const unsigned long BlockOffset[] =
{
   0x000000L,  /* Start offset of block 70 */
   0x008000L,  /* Start offset of block 69 */
   0x010000L,  /* Start offset of block 68 */
   0x018000L,  /* Start offset of block 67 */
   0x020000L,  /* Start offset of block 66 */
   0x028000L,  /* Start offset of block 65 */
   0x030000L,  /* Start offset of block 64 */
   0x038000L,  /* Start offset of block 63 */
   0x040000L,  /* Start offset of block 62 */
   0x048000L,  /* Start offset of block 61 */
   0x050000L,  /* Start offset of block 60 */
   0x058000L,  /* Start offset of block 59 */
   0x060000L,  /* Start offset of block 58 */
   0x068000L,  /* Start offset of block 57 */
   0x070000L,  /* Start offset of block 56 */
```

```
   0x078000L,  /* Start offset of block 55 */
   0x080000L,  /* Start offset of block 54 */
   0x088000L,  /* Start offset of block 53 */
   0x090000L,  /* Start offset of block 52 */
   0x098000L,  /* Start offset of block 51 */
   0x0A0000L,  /* Start offset of block 50 */
   0x0A8000L,  /* Start offset of block 49 */
   0x0B0000L,  /* Start offset of block 48 */
   0x0B8000L,  /* Start offset of block 47 */
   0x0C0000L,  /* Start offset of block 46 */
   0x0C8000L,  /* Start offset of block 45 */
   0x0D0000L,  /* Start offset of block 44 */
   0x0D8000L,  /* Start offset of block 43 */
   0x0E0000L,  /* Start offset of block 42 */
   0x0E8000L,  /* Start offset of block 41 */
   0x0F0000L,  /* Start offset of block 40 */
   0x0F8000L,  /* Start offset of block 39 */
   0xF00000L,  /* Start offset of block 38 */
   0x108000L,  /* Start offset of block 37 */
   0x110000L,  /* Start offset of block 36 */
   0x118000L,  /* Start offset of block 35 */
   0x120000L,  /* Start offset of block 34 */
   0x128000L,  /* Start offset of block 33 */
   0x130000L,  /* Start offset of block 32 */
   0x138000L,  /* Start offset of block 31 */
   0x140000L,  /* Start offset of block 30 */
   0x148000L,  /* Start offset of block 29 */
   0x150000L,  /* Start offset of block 28 */
   0x158000L,  /* Start offset of block 27 */
   0x160000L,  /* Start offset of block 26 */
   0x168000L,  /* Start offset of block 25 */
   0x170000L,  /* Start offset of block 24 */
   0x178000L,  /* Start offset of block 23 */
   0x180000L,  /* Start offset of block 22 */
   0x188000L,  /* Start offset of block 21 */
   0x190000L,  /* Start offset of block 20 */
   0x198000L,  /* Start offset of block 19 */
   0x1A0000L,  /* Start offset of block 18 */
   0x1A8000L,  /* Start offset of block 17 */
   0x1B0000L,  /* Start offset of block 16 */
   0x1B8000L,  /* Start offset of block 15 */
   0x1C0000L,  /* Start offset of block 14 */
   0x1C8000L,  /* Start offset of block 13 */
   0x1D0000L,  /* Start offset of block 12 */
   0x1D8000L,  /* Start offset of block 11 */
   0x1E0000L,  /* Start offset of block 10 */
   0x1E8000L,  /* Start offset of block 9 */
   0x1F0000L,  /* Start offset of block 8 */
   0x1F8000L,  /* Start offset of block 7 */
   0x1F9000L,  /* Start offset of block 6 */
   0x1FA000L,  /* Start offset of block 5 */
   0x1FB000L,  /* Start offset of block 4 */
   0x1FC000L,  /* Start offset of block 3 */
   0x1FD000L,  /* Start offset of block 2 */
   0x1FE000L,  /* Start offset of block 1 */
   0x1FF000L   /* Start offset of block 0 */
};
#endif /* USE_M28W320CT */
```

```
#ifdef USE_M28W320CB
/* Block organisation for Bottom Boot Block devices */
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 */
};
#endif /* USE_M28W320CB */

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

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

The following function is only needed in this module.
*****************************************************************************/
static unsigned int FlashWrite( unsigned long ulOff, unsigned int uVal );

/*****************************************************************************
Function:    unsigned int FlashWrite( unsigned long ulOff, unsigned int uVal )
Arguments:   ulOff is 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 FlashReadReset( void )
Arguments:     none
Return Value:  none
Description:   This function places the flash in the Read Array 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 Array mode so this is
   not strictly necessary. It is provided for completeness.

Pseudo Code:
   Step 1: write command sequence (see Instructions Table of the Data Sheet)
*******************************************************************************/
void FlashReadReset( void )
{
   /* Step 1: write command sequence */
   FlashWrite( ANY_ADDR, 0x00FF );
}

/*******************************************************************************
Function:      int FlashAutoSelect( int iFunc )
Arguments:     iFunc should be set to one of the Read Signature values. The
   header file defines the values for reading the Signature.
Return Value:  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:
      M28W800CT  88CCh
      M28W800CB  88CDh
      M28W160CT   88CEh
      M28W160CB   88CFh
      M28W320CT  88BAh
      M28W320CB  88BBh
   When iFunc is invalid the function returns FLASH_FUNCTION_NOT_SUPPORTED (-12)
Description:   This function can be used to read the electronic signature of the
   device or the manufacturer code.
Note: The command sequence given here is not described in the Data Sheet. The
   addresses and commands are applicable to the M29 series FLASH devices.
   The only command required for the M28WxxxC is the final one, the 0x0090
   command (Read Electronic Signature Command). The other writes to the command
   interface will be ignored by the M28WxxxC. The advantage of this is that the
   FlashAutoSelect() function will be able to identify the M29 series Flash
   devices AS WELL AS the M28 series Flash devices.

   Unlike many other Flash Memories, it is important to ensure A1 to A7 are
```

```
   low when reading the manufacturer and device codes as M28WxxxC uses bits
   A1 to A7 to access the Protection Register.

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

   /* Step 1: Send the Read Electronic Signature instruction */
   FlashWrite( 0x5555L, 0x00AA ); /* 1st Cycle, ignored by M28 series devices */
   FlashWrite( 0x2AAAL, 0x0055 ); /* 2nd Cycle, ignored by M28 series devices */
   FlashWrite( 0x5555L, 0x0090 ); /* 3rd Cycle */

   /* Step 2: Read the required function */
   if( iFunc == FLASH_READ_MANUFACTURER )
      iRetVal = FlashRead( 0x0000L ); /* A0 = A1 = 0 */

   else if( iFunc == FLASH_READ_DEVICE_CODE )
      iRetVal = 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 Array mode */
   FlashWrite( ANY_ADDR, 0x00FF );

   return iRetVal;
}

/*****************************************************************************
Function:      int FlashReadCFI( int iCFIFunc, unsigned int *uCFIValue )
Arguments:     iCFIFunc is set to the offset of the CFI parameter to be read.
   The CFI value read from offset iCFIFunc is passed back to the calling
   function by *uCFIValue.

Return Value: On success returns FLASH_SUCCESS (-1) or if the CFI cannot be
   identified by reading the chracters QRY from locations 0x10, 0x11 and 0x12,
   the function returns FLASH_CFI_FAIL (-19)
Description: This function checks that the flash CFI is present and operable,
   then reads the CFI value at the specified offset. The CFI value requested is
   then passed back to the calling function.
Note: Pseudo Code:
   Step 1: Send the Read CFI Instruction
   Step 2: Check that the CFI interface is operable
   Step 3: If CFI is operable read the required CFI value
   Step 4: Return the flash to Read Array mode
*****************************************************************************/
int FlashReadCFI( int iCFIFunc, unsigned int *uCFIValue )
{
   int iRetVal = FLASH_SUCCESS; /* Holds the return value */
   unsigned long ulCFIAddr;     /* Holds CFI address */
```

```
   /* Step 1: Send the Read CFI Instruction */
   FlashWrite( 0x0055L, 0x0098 );

   /* Step 2: Check that the CFI interface is operable */
   if( (FlashRead( 0x0010L ) != 'Q' ) || (FlashRead( 0x0011L ) !='R' )
   || (FlashRead( 0x0012L ) != 'Y' ))
      iRetVal=FLASH_CFI_FAIL;

   else
   {
      /* Step 3: Read the required CFI */
      ulCFIAddr = (unsigned long)iCFIFunc;
      *uCFIValue = FlashRead( ulCFIAddr & 0x00FFL );
   }

   /* Step 4: Return to Read Array mode */
   FlashWrite( ANY_ADDR, 0x00FF );

   return iRetVal;
}

/*****************************************************************************
Function:     int FlashBlockErase( unsigned char ucBlock )
Arguments:    ucBlock is the number of the Block to be erased.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS             (-1)
   FLASH_WRONG_TYPE          (-8)
   FLASH_BLOCK_INVALID       (-5)
   FLASH_PROTECTED           (-11)
   FLASH_VPP_INVALID         (-13)
   FLASH_BLOCK_FAILED_ERASE  (-9)
Description:  This function can be used to erase the Block specified in ucBlock.
   The function checks that the block is valid before issuing the erase
   command. Once the erase has completed the function checks the Status
   Register for errors. Any errors are returned, otherwise FLASH_SUCCESS
   is returned.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check that the block is valid
   Step 3:  Issue Erase Command
   Step 4:  Wait until Program/Erase Controller is ready
   Step 5:  Check for any errors
   Step 6:  Return to Read Array mode
   Step 7:  Return error condition
*****************************************************************************/
int FlashBlockErase( unsigned char ucBlock )
{
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
   unsigned int uStatus;        /* Holds the Status Register reads */

   /* 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 the block is valid */
   if( ucBlock >= NUM_BLOCKS )
      return FLASH_BLOCK_INVALID;
```

```
   /* Step 3: Issue Erase Command */
   FlashWrite( ANY_ADDR, 0x0050 ); /* Clear Status Register */

   /* NOTE ! CSR also clears b1 BPS as well as b3,4 and 5 */

   FlashWrite( ANY_ADDR, 0x0020 );             /* 1st cycle */
   FlashWrite( BlockOffset[ucBlock], 0x00D0 ); /* 2nd cycle */

   /* Step 4:    Wait until Program/Erase Controller is ready */
   /* TimeOut! */
   do
      uStatus = FlashRead(ANY_ADDR);
   while( (uStatus&0x0080) == 0x0000 );

   /* Step 5: Check for any errors */
   if( uStatus&0x0002 )
      iRetVal = FLASH_PROTECTED;
   else if( uStatus&0x0008 )
      iRetVal = FLASH_VPP_INVALID;
   else if( uStatus&0x0020 )
      iRetVal = FLASH_BLOCK_FAILED_ERASE;

   /* Step 6:  Return to Read Array mode */
   FlashWrite( ANY_ADDR, 0x0050 );            /* Clear Status Register */

   /* NOTE ! CSR also clears b1 BPS as well as b3,4 and 5 */

   FlashWrite( ANY_ADDR, 0x00FF );            /* Read Array Command */

   /* Step 7: Return error condition */
   return iRetVal;
}

/*****************************************************************************
Function:     int FlashChipErase( int *iResults )
Arguments:    iResults is a pointer to an array where the results will be
   stored. If iResults == NULL then no results are stored.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS      (-1)
   FLASH_WRONG_TYPE   (-8)
   FLASH_ERASE_FAIL   (-14)
   If FLASH_SUCCESS is returned then Results is left unchanged.
   If FLASH_ERASE_FAIL is returned then Results will be filled with the error
   conditions for each block. The possible error conditions are:
   FLASH_SUCCESS      (-1)
   FLASH_PROTECTED    (-11)
   FLASH_VPP_INVALID  (-13)
   FLASH_ERASE_FAIL   (-14)
Description: The function can be used to erase the whole flash chip. Each Block
   is erased in turn. The function only returns when all of the Blocks have
   been erased or have generated an error, except if the FLASH_VPP_INVALID is
   encountered, in which case the function aborts and reports all remaining
   blocks as having FLASH_VPP_INVALID. If Vpp is invalid for one block then it
   follows that it will be invalid for subsequent blocks (battery failure?).

Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: For each block
```

```
   Step 3: Send Block Erase Command
   Step 4: Register the errors in the array
   Step 5: If FLASH_VPP_INVALID returned fill rest of results array & abort
   Step 6: Return error condition
*****************************************************************************/
int FlashChipErase( int *iResults )
{
   unsigned char ucCurBlock;    /* Used to track the current block in a range */
   int iRetVal = FLASH_SUCCESS; /* Return value: Initially optimistic */
   int iError;                  /* Holds the latest error */

   /* 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: For each block */
   for( ucCurBlock = 0; ucCurBlock < NUM_BLOCKS; ucCurBlock++ )
   {
      /* Step 3: Send Block Erase Command */
      iError = FlashBlockErase( ucCurBlock );
      if( iError != FLASH_SUCCESS )
         iRetVal = FLASH_ERASE_FAIL;

      /* Step 4: Register the errors in the array */
      if( iResults != NULL )
         iResults[ucCurBlock] = iError;

      /* Step 5: If FLASH_VPP_INVALID returned fill rest of results array
                 & abort */
      if( iError == FLASH_VPP_INVALID )
      {
         if( iResults != NULL )
            while( ++ucCurBlock < NUM_BLOCKS ) /* on remaining blocks */
               iResults[ucCurBlock] = iError;  /* fill in Vpp error */
         /* the for() loop will now terminate since ucCurBlock == NUM_BLOCKS */
      }
   }

   /* Step 6: Return error condition */
   return iRetVal;
}

/*****************************************************************************
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 is the block really needs unprotecting as it may already be
            unprotected
   Step 4:  Unprotect the block
   Step 5:  Check the block ucBlock for protection status
   Step 6:  Check for errors
*****************************************************************************/
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( (int)ucBlock );

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

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

   /* Step 4 Unprotect the block */
   FlashWrite(ANY_ADDR,0x0060);
   FlashWrite(BlockOffset[ucBlock], 0x00D0);

   /* Step 5 Check block ucBlock for protection status */
   iRetVal = FlashAutoSelect( (int)ucBlock );

   /* Step 6: Check for any errors */
   if ((iRetVal == FLASH_BLOCK_PROTECTED)
   ||  (iRetVal == FLASH_BLOCK_LOCKED_PROTECTED))
      return FLASH_UNPROTECT_FAIL;
   else
      return FLASH_SUCCESS;
}

/*****************************************************************************
Function:     int FlashBlockProtect(unsigned char ucBlock)
Arguments:    ucBlock holds the block number to protect
Return Value: The function returns the following conditons:
   FLASH_WRONG_TYPE     (-8)
   FLASH_SUCCESS        (-1)
   FLASH_PROTECT_FAIL   (-18)
   FLASH_PROTECTED      (-11)
   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 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 is the block really needs unprotecting as it may already be
            unprotected
   Step 4:  Unprotect the block
   Step 5:  Read the block ucBlock to check protection status
   Step 6:  Check for error condition
```

```
*******************************************************************************/
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( (int)ucBlock );

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

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

   /* Step 4 Protect the block */
   FlashWrite(ANY_ADDR,0x0060);
   FlashWrite(BlockOffset[ucBlock], 0x0001);

   /* Step 5 Check block ucBlock for protection status */
   iRetVal = FlashAutoSelect( (int)ucBlock );

   /* Step 6: Check for any errors */
   if (iRetVal < FLASH_BLOCK_PROTECTED)
      return FLASH_PROTECT_FAIL;
   else
      return FLASH_SUCCESS;
}

/*******************************************************************************
Function:     int FlashBlockLock(unsigned char ucBlock)
Arguments:    ucBlock holds the block number to lock
Return Value: The function returns the following conditons:
   FLASH_LOCKED        (-20) On success
   FLASH_LOCK_FAIL     (-21) If the locking was unsuccessful
   FLASH_BLOCK_INVALID (-5)  If the block is not valid

Description:   This function locks a block selected by ucBlock but only if
   that particular block is valid. It then tests to see if the lock command 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 is the block really needs locking as it may already be
            locked
   Step 4:  Lock the block
   Step 5:  Check that it locked
*******************************************************************************/
int FlashBlockLock(unsigned char ucBlock)
{
   /* 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: Is the block valid ? */
   if (FlashAutoSelect( (int)ucBlock ) == FLASH_BLOCK_INVALID)
      return FLASH_BLOCK_INVALID;

   /* Step 3: Do not lock if the block is already locked */
   if (((FlashAutoSelect( (int)ucBlock )) & 0x02) == FLASH_BLOCK_LOCKED)
      return FLASH_LOCKED;

   /* Step 4: Lock the block */
   FlashWrite(ANY_ADDR,0x0060);
   FlashWrite(BlockOffset[ucBlock], 0x002F);

   /* Step 5: Check that it locked */
   if (((FlashAutoSelect( (int)ucBlock )) & 0x02) == FLASH_BLOCK_LOCKED)
      return FLASH_SUCCESS;
   else
      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:  On success the function returns FLASH_SUCCESS (-1).
   If a protected address is given the function returns FLASH_PROTECTED (-11).
   If Vpp is invalid then the function returns FLASH_VPP_INVALID (-13)
   On failure the function returns FLASH_PROGRAM_FAIL (-6).
   If the address range to be programmed exceeds the address range of the Flash
   Device the function returns FLASH_ADDRESS_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.
Description: This function is used to program an array into the flash. It does
   not erase the flash first and will 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: While there is more to be programmed
   Step 4: Program the next word
   Step 5: Wait until the Program/Erase Controller is ready
   Step 6: Check for any errors
   Step 7: Update pointers
   Step 8: Step 8: Clear status register and return to read array mode
   Step 9: Return the error condition
*****************************************************************************/
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 int uStatus;        /* Holds the Status Register reads */
   int iRetVal = FLASH_SUCCESS; /* Return Value: Initially optimistic */

   /* Step 1: Check that the flash is of the correct type */
   if( !(FlashAutoSelect( FLASH_READ_MANUFACTURER ) == MANUFACTURER_ST)
   ||  !(FlashAutoSelect( FLASH_READ_DEVICE_CODE ) == EXPECTED_DEVICE ) )
```

```
      return FLASH_WRONG_TYPE;

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

   /* Step 3: While there is more to be programmed */
   uArrayPointer = (unsigned int *)Array;
   while( ulOff <= ulLastOff && iRetVal == FLASH_SUCCESS )
   {
      /* Step 4: Program the next word */
      FlashWrite( ANY_ADDR, 0x0050 );        /* Clear Status Register */
      FlashWrite( ANY_ADDR, 0x0040 );        /* Program Set-up */
      FlashWrite( ulOff, *uArrayPointer );   /* Program value */

      /* Step 5: Wait until Program/Erase Controller is ready */
      /* TimeOut! */
      do
         uStatus = FlashRead(ANY_ADDR);
      while( (uStatus&0x0080) == 0x0000 );

      /* Step 6: Check for any errors */
      if( uStatus&0x0002 )
         iRetVal = FLASH_PROTECTED;
      else if( uStatus&0x0008 )
         iRetVal = FLASH_VPP_INVALID;
      else if( uStatus&0x0010 )
         iRetVal = FLASH_PROGRAM_FAIL;

      /* Step 7: Update pointers */
      ulOff++;          /* next word offset */
      uArrayPointer++;  /* next word in array */
   }

   /* Step 8: Clear status register and return to read array mode */
   FlashWrite( ANY_ADDR, 0x0050 ); /* Clear Status Register */

   /* NOTE ! CSR also clears b1 BPS as well as b3,4 and 5 */

   FlashWrite( ANY_ADDR, 0x00FF ); /* Read Array Command */

   /* Step 9: return the error condition */
   return iRetVal;
}

/*****************************************************************************
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 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 CFI Query Failed",
                          "Flash Locked",
                          "Flash Lock Failed"
                        };

   /* 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:        Applies to M29 series FLASH only. 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. This error condition should not
   occur when using this library.
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. This error condition should not
   occur when using this library.
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 have blocks protected. It is also possible for the
   block to be locked but is unlikely if /WP pin is kept high.
   If the program fail still results it is likely that the flash is suspect.
*******************************************************************************

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 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:     The block may be protected/locked or the flash is suspect.
```

```
*******************************************************************************

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

Error Name:  FLASH_PROTECTED
Return Value: -11
Description:  The user has attempted to erase, program or protect a block of
   the flash that is protected. The operation failed because the block was
   protected.
Solutions:    Choose another (unprotected) block for erasing or programming.
   Alternatively change the block protection status of the current block.
   (see Datasheet for more details). In the case of the user protecting a block
   that is already protected, this warning notifies the user that the command
   had no effect.
*******************************************************************************

Return Name:  FLASH_FUNCTION_NOT_SUPPORTED
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.
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 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:        Applies to M29 series Flash only. This error condition should not
   occur when using this library.
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
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
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_CFI_FAIL
Return Value: -19
Description:  This error return value indicates that a CFI read was unsuccessful

Solution:     Try an Autoselect command if this fails it is likely that the
   device is faulty or the interface to the flash is not correct.
*********************************************************************************

Error Name:   Flash_LOCKED
Return Value: -20
Description:  This error return value indicates that the flash block selected
   was already locked. This is not an error but a method of informing the user
   that the command had no effect.

Solution: The block was already locked. Was the right block selected ?
*********************************************************************************

Error Name:   Flash_LOCK_FAIL
Return Value: -21
Description:  This error return value indicates that the flash lock command on
   the selected block failed.

Solution:  Try an Autoselect command if this fails it is likely that the device
   is faulty or the interface to the flash is not correct.
*********************************************************************************/
```

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.