AN1270
APPLICATION NOTE

Software Drivers for the M58LW064A Flash Memory

CONTENTS

INTRODUCTION

This application note provides library source code in C for the M58LW064A Multi-Bit Cell Flash Memory. The M58LW064A supports both Asynchronous and Synchronous Burst bus interfaces. The C code drivers work on a layer above the hardware and can be used successfully over either bus interface.

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 c1270_16.c and c1270_16.h files contain libraries for accessing the M58LW064A Flash Memories.

Also included in this application note is an overview of the programming model for the M58LW064A. 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 M58LW064A 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 tested on a target platform. It is small in size and can be applied to any target hardware.

THE M58LW064A PROGRAMMING MODEL

The M58LW064A is a 64Mb (4Mb x16) Flash Memory which can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is broken down into 64 blocks, each 64 Kwords in size. Each block can be erased individually.

The M58LW064A is a smart voltage device. It differs from first generation dual voltage devices which require a 12V supply to program or erase. The M58LW064A 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. Two power supply

pins are used to give the optimal supply voltage conditions. In normal operation the core (V_{DD} pin) is supplied with 3V and the I/O buffers (V_{DDQ}) are supplied with 1.8V to 3V.

Unlike many other Flash memories, the V_{PP} Program/Erase Enable pin is not a supply pin; instead it operates using logic levels. Traditionally a jumper or a high power circuit was required to drive this type of pin. On the M58LW064A the pin can be driven directly from a logic buffer, enabling easy control of global block protection by the microprocessor.

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 M58LW064A 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 (150ns part).

Bus Operations and Commands

Most of the functionality of the M58LW064A 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 M58LW064. The various commands recognized by the M58LW064 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 (Write to Buffer and Program)
5. Program/Erase Suspend
6. Common Flash Interface Query
7. Block Protect/Blocks Unprotect
8. Asynchronous/Synchronous bus interface configuration.

The Read command returns the M58LW064A to Read mode 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 while in the Auto Select mode. In the library of software drivers this mode is known as the Auto Select mode to make the M58LW064A 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 block while the memory is protected (e.g. $V_{PP} = V_{IL}$) generate an error and do not modify the contents of the memory.

The Write to Buffer and Program command is used to modify the data stored at the specified addresses of the device. Programming modifies one or more pages (4 words) at a time. Unlike previous Flash memories, which are able to program each word or byte at a time, the M58LW064A must be programmed one page at a time. Once any word within the page is programmed it is no longer possible to change it, or any other word within that page, until an Erase command has been issued on the whole block. Each Write to Buffer and Program command can program up to 4 pages (16 Words); programming larger amounts of

data must be performed one buffer at a time, by issuing a Write to Buffer and Program command, waiting for the command to complete, then issuing the next Write to Buffer and Program command, and so on.

Issuing the Program/Erase Suspend command during a Program or Erase operation will temporarily place the M58LW064A 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 192µs for programming and 0.75s for erasing on the M58LW064A. 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 on the M58LW064A is non-volatile; after power up or a hardware reset the block protection remains at the state it was in before the power-down or reset. Each Block can be protected individually, or all Blocks can be unprotected at the same time; it is not possible to unprotect one block, leaving any others protected.

The bus configuration of the device can be changed. The device supports both asynchronous and synchronous bus operations, with many different options for burst synchronous reads and asynchronous page reads. Note that some modes of the M58LW064A do not support synchronous read operations, when one of these modes is selected the part will automatically revert to asynchronous mode and return to synchronous mode afterwards.

The Status Register

While the M58LW064A 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 M58LW064A 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 Commands Table of the M58LW064A 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) = 0x00E8; /* 1st cycle (any block address) */
*(unsigned int*)(0x0000) = 0x0000; /* 2nd cycle: data length */
*(unsigned int*)(0x03E2) = 0x9465; /* 3rd cycle: address and data */
*(unsigned int*)(0x0000) = 0x00D0; /* final cycle: confirm and start */
```

The first two addresses (0000h) are arbitrary, so long as they are inside the block where the data is to be programmed. This example assumes that address 0000h of the M58LW064A 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 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.

Following the programming, address 03E2h cannot be modified until an Erase operation has been used to erase the block. Furthermore it will not be possible to program any of the addresses in the range 03E0h to 03E3h until an Erase operation has been used to erase the block; any attempt to do so will generate an error.

WRITING C CODE FOR THE M58LW064A

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 (M58LW064A). 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 M58LE064A. 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 M58LW064A 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.

It should be noted that, if upgrading from a previous Flash memory then all calls to the function FlashProgram() in the application will have to be checked. Previous Flash were able to program each word/byte individually; this technique can no longer be used and it is necessary to program page by page.

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 Version Code 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.

`FlashBlockErase()` is used to erase a block in the device. The blocks cannot be erased when V_{PP} is Low, V_{IL}: attempting to do so generates an error.

`FlashChipErase()` is used to erase the entire chip. The chip cannot be erased when V_{PP} is Low, V_{IL}: attempting to do so generates an error.

`FlashProgram()` is used to program data arrays into the Flash. Only previously erased double words can be programmed reliably. Again, programming cannot take place when V_{PP} Low, V_{IL}.

`FlashBlockProtect()` is used to protect a block in the Flash. Once protected, the data in the block cannot be programmed or erased until the block is unprotected.

`FlashChipUnprotect()` is used to unprotect all the blocks in the Flash memory. Once unprotected, all the data in all the blocks can be erased or programmed.

`FlashSetBurstConfig()` is used to change the bus configuration and select asynchronous or synchronous modes. To help the programmer select the correct settings, the header file contains many bit definitions that can be used.

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 long FlashRead( unsigned long *Addr);
unsigned long 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 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 M58LW064A. 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:

When the device is in Read mode interrupts can freely read from the device.

Interrupts which do not access the device may be used during all the functions.

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.

LIMITATIONS OF THE SOFTWARE

The software provided does not implement a full set of the M58LW064A's functionality. It is left to the user to implement the Program/Erase Suspend command 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 M58LW064A 3V Flash Memory is an ideal product for 16 bit 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	Revision Details
July 2001	First issue
September 2001	Tested using new silicon

```
/**** c1270_16.h Header File for Flash Memory ********************************

   Filename:    c1270_16.h
   Description: Header file for c1270_16.c. Consult the C file for details.

   Copyright (c) 2001 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
******************************************************************************/
/* Read Signature values */
#define FLASH_READ_MANUFACTURER    (-1)
#define FLASH_READ_DEVICE_CODE     (-2)

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

See end of C file for explanations and help
******************************************************************************/
#define FLASH_BLOCK_PROTECTED         (0x0001)
#define FLASH_BLOCK_UNPROTECTED       (0x0000)

#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_VPP_INVALID             (-13)
#define FLASH_ERASE_FAIL              (-14)
#define FLASH_UNPROTECT_FAIL          (-16)
#define FLASH_PROTECT_FAIL            (-18)
#define FLASH_CFI_FAIL                (-22)
#define FLASH_CONFIG_INVALID          (-23)

/******************************************************************************
Flash Burst Configuration Definitions
******************************************************************************/
/* Burst Configuration Register Read Mode */
#define FLASH_BCR_SYNCHRONOUS    0x00000000L
#define FLASH_BCR_ASYNCHRONOUS   0x00008000L
/* Burst Configuration Register X-latency */
#define FLASH_BCR_XLAT_7         0x00001000L
#define FLASH_BCR_XLAT_8         0x00001800L
#define FLASH_BCR_XLAT_9         0x00002000L
#define FLASH_BCR_XLAT_10        0x00002800L
#define FLASH_BCR_XLAT_11        0x00003000L
#define FLASH_BCR_XLAT_12        0x00004800L
#define FLASH_BCR_XLAT_13        0x00005000L
```

```
#define FLASH_BCR_XLAT_15        0x00006800L
/* Burst Configuration Register Y-latency */
#define FLASH_BCR_YLAT_1         0x00000000L
#define FLASH_BCR_YLAT_2         0x00000200L
/* Burst Configuration Register valid Data Ready timing */
#define FLASH_BCR_DR_NORMAL      0x00000000L
#define FLASH_BCR_DR_ADVANCE     0x00000100L
/* Burst Configuration Register Burst Type */
#define FLASH_BCR_INTERLEAVE     0x00000000L
#define FLASH_BCR_SEQUENTIAL     0x00000080L
/* Burst Configuration Register valid Clock Edge */
#define FLASH_BCR_CLK_FALLING    0x00000000L
#define FLASH_BCR_CLK_RISING     0x00000040L
/* Burst Configuration Register Latch Enable activation */
#define FLASH_BCR_L_DISABLED     0x00000000L
#define FLASH_BCR_L_ENABLED      0x00000008L
/* Burst Configuration Register Burst Length */
#define FLASH_BCR_LENGTH_1       0x00000004L
#define FLASH_BCR_LENGTH_2       0x00000005L
#define FLASH_BCR_LENGTH_4       0x00000001L
#define FLASH_BCR_LENGTH_8       0x00000002L
#define FLASH_BCR_CONTINUOUS     0x00000007L

/*****************************************************************************
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 FlashProgram( unsigned long ulOff, size_t NumWords, void *Array );
extern int FlashBlockProtect( unsigned char ucBlock );
extern int FlashChipUnprotect( void );
extern int FlashSetBurstConfig( unsigned long ulBurstConfig );
extern char *FlashErrorStr( int iErrNum );
```

```
/**** c1270_16.c  M58LW064 Flash Memory Driver ********************************

   Filename:    c1270_16.c
   Description: Library routines for the M58LW064 64Mb (4Mb x16) Flash Memory.
                16 bit drivers.

   Version:     1.02
   Date:        07/09/2001
   Author:      Tim Webster & Alex Nairac, Oxford Technical Solutions

   Copyright (c) 2001 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   13/06/2001  Initial release of the software (untested).
   1.01   22/06/2001  Most functions tested on M58LW064A.
                      Functions FlashBlockUnprotect(), FlashChipProtect() and
                      FlashSetBurstConfig() not yet tested.
   1.02   07/09/2001  Re-test using final silicon version. All functions OK.

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

   This source file provides library C code for using the M58LW064 devices.
   The following devices are supported in the code:
      M58LW064A
   This file is used to access the devices in 16-bit mode only.

   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
      FlashProgram()            to program a word or an array
      FlashBlockProtect()       to protect a block of the memory
      FlashChipUnprotect()      to unprotect the whole memory
      FlashSetBurstConfig()     to set the burst configuration register
      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 “c1270_16.h”   /* Header file with global prototypes */

#define USE_M58LW064A

/*****************************************************************************
Constants
*****************************************************************************/
#define MANUFACTURER_ST (0x0020)  /* ST manufacturer code is 0x20 */
#define BASE_ADDR ((volatile unsigned int*)0x00000000L)
   /* 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 (0x00000000L)
   /* Any address offset within the Flash Memory will do */

#ifdef  USE_M58LW064A
#define EXPECTED_DEVICE (0x0017)   /* Device code for the M58LW064A */
#define FLASH_SIZE (0x00400000L)   /* Total device size */

static const unsigned long BlockOffset[] =
{
   0x00000000L,  /* Start offset of block 0  */
   0x00010000L,  /* Start offset of block 1  */
   0x00020000L,  /* Start offset of block 2  */
   0x00030000L,  /* Start offset of block 3  */
   0x00040000L,  /* Start offset of block 4  */
   0x00050000L,  /* Start offset of block 5  */
   0x00060000L,  /* Start offset of block 6  */
   0x00070000L,  /* Start offset of block 7  */
   0x00080000L,  /* Start offset of block 8  */
   0x00090000L,  /* Start offset of block 9  */
   0x000A0000L,  /* Start offset of block 10 */
   0x000B0000L,  /* Start offset of block 11 */
   0x000C0000L,  /* Start offset of block 12 */
   0x000D0000L,  /* Start offset of block 13 */
   0x000E0000L,  /* Start offset of block 14 */
   0x000F0000L,  /* Start offset of block 15 */
   0x00100000L,  /* Start offset of block 16 */
   0x00110000L,  /* Start offset of block 17 */
   0x00120000L,  /* Start offset of block 18 */
```

```
   0x00130000L,  /* Start offset of block 19 */
   0x00140000L,  /* Start offset of block 20 */
   0x00150000L,  /* Start offset of block 21 */
   0x00160000L,  /* Start offset of block 22 */
   0x00170000L,  /* Start offset of block 23 */
   0x00180000L,  /* Start offset of block 24 */
   0x00190000L,  /* Start offset of block 25 */
   0x001A0000L,  /* Start offset of block 26 */
   0x001B0000L,  /* Start offset of block 27 */
   0x001C0000L,  /* Start offset of block 28 */
   0x001D0000L,  /* Start offset of block 29 */
   0x001E0000L,  /* Start offset of block 30 */
   0x001F0000L,  /* Start offset of block 31 */
   0x00200000L,  /* Start offset of block 32 */
   0x00210000L,  /* Start offset of block 33 */
   0x00220000L,  /* Start offset of block 34 */
   0x00230000L,  /* Start offset of block 35 */
   0x00240000L,  /* Start offset of block 36 */
   0x00250000L,  /* Start offset of block 37 */
   0x00260000L,  /* Start offset of block 38 */
   0x00270000L,  /* Start offset of block 39 */
   0x00280000L,  /* Start offset of block 40 */
   0x00290000L,  /* Start offset of block 41 */
   0x002A0000L,  /* Start offset of block 42 */
   0x002B0000L,  /* Start offset of block 43 */
   0x002C0000L,  /* Start offset of block 44 */
   0x002D0000L,  /* Start offset of block 45 */
   0x002E0000L,  /* Start offset of block 46 */
   0x002F0000L,  /* Start offset of block 47 */
   0x00300000L,  /* Start offset of block 48 */
   0x00310000L,  /* Start offset of block 49 */
   0x00320000L,  /* Start offset of block 50 */
   0x00330000L,  /* Start offset of block 51 */
   0x00340000L,  /* Start offset of block 52 */
   0x00350000L,  /* Start offset of block 53 */
   0x00360000L,  /* Start offset of block 54 */
   0x00370000L,  /* Start offset of block 55 */
   0x00380000L,  /* Start offset of block 56 */
   0x00390000L,  /* Start offset of block 57 */
   0x003A0000L,  /* Start offset of block 58 */
   0x003B0000L,  /* Start offset of block 59 */
   0x003C0000L,  /* Start offset of block 60 */
   0x003D0000L,  /* Start offset of block 61 */
   0x003E0000L,  /* Start offset of block 62 */
   0x003F0000L,  /* Start offset of block 63 */
};
#endif /* USE_M58LW064A */
#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))

/* Burst Configuration Register definitions */
#define FLASH_BCR_MODE_MASK    0x00008000L
#define FLASH_BCR_XLAT_MASK    0x00007800L
#define FLASH_BCR_YLAT_MASK    0x00000200L
#define FLASH_BCR_LENGTH_MASK  0x00000007L

/*****************************************************************************
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 double-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 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 either to the Read Signature values or to the
   block number. The header file defines the values for reading the Signature.
   Note: the first block is Block 0
Return Value:  When iFunc is >= 0 the function returns FLASH_BLOCK_PROTECTED
   (01h) if the block is protected and FLASH_BLOCK_UNPROTECTED (00h) if it is
   unprotected. See the Auto Select command in the Data Sheet for further
   information.

   When iFunc is FLASH_READ_MANUFACTURER (-2) the function returns the
   manufacturer's code. The Manufacturer code for ST is 0020h.

   When iFunc is FLASH_READ_DEVICE_CODE (-1) the function returns the Device
   Code.  The device codes for the parts are:
      M58LW064A   0x0017

   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 or the manufacturer code. It can also be used to read the protection
   status of a block.

Pseudo Code:
   Step 1:  Send the Read Electronic Signature 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( ANY_ADDR, 0x0090 );

   /* Step 2: Read the required function */
   if( iFunc == FLASH_READ_MANUFACTURER )
      iRetVal = FlashRead( 0x00000000L ); /* A1 = A2 = 0 */

   else if( iFunc == FLASH_READ_DEVICE_CODE )
      iRetVal = FlashRead( 0x00000001L ); /* A1 = 1, A2 = 0 */

   else if( (iFunc >= 0) && (iFunc < NUM_BLOCKS) )
      iRetVal = FlashRead( BlockOffset[iFunc] + 0x00000002L );
                                          /* A1 = 0, A2 = 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 (-22)
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.

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( ANY_ADDR, 0x0098 );

   /* Step 2: Check that the CFI interface is operable */
   if( ((FlashRead( 0x00000010L ) & 0x00FF) != 0x0051) ||
       ((FlashRead( 0x00000011L ) & 0x00FF) != 0x0052) ||
       ((FlashRead( 0x00000012L ) & 0x00FF) != 0x0059) )
      iRetVal=FLASH_CFI_FAIL;
   else
   {
      /* Step 3: Read the required CFI */
      ulCFIAddr = (unsigned long)iCFIFunc;
      *uCFIValue = FlashRead( ulCFIAddr & 0x000000FFL );
   }

   /* 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_BLOCK_INVALID        (-5)
   FLASH_WRONG_TYPE           (-8)
   FLASH_BLOCK_FAILED_ERASE   (-9)
   FLASH_PROTECTED            (-11)
   FLASH_VPP_INVALID          (-13)
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:  Check to see if the block is protected
   Step 4:  Issue Erase Command
   Step 5:  Wait until Program/Erase Controller is ready
   Step 6:  Check for any errors
   Step 7:  Clear Status Register and return to Read Array mode
   Step 8:  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: Check to see if the block is protected */
   if ( FlashAutoSelect( (int)ucBlock ) == FLASH_BLOCK_PROTECTED)
      return FLASH_PROTECTED;

   /* Step 4: Issue Erase Command */
   FlashWrite( ANY_ADDR, 0x0050L ); /* Clear Status Register */
   /* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */
   FlashWrite( ANY_ADDR, 0x0020 );             /* 1st cycle */
   FlashWrite( BlockOffset[ucBlock], 0x00D0 ); /* 2nd cycle */

   /* 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 & 0x0008 )
      iRetVal = FLASH_VPP_INVALID;
   else if( uStatus & 0x0020 )
      iRetVal = FLASH_BLOCK_FAILED_ERASE;
   else if( uStatus & 0x0002 )
      iRetVal = FLASH_PROTECTED;
   else if( uStatus & 0x0010 )
      iRetVal = FLASH_BLOCK_FAILED_ERASE;

   /* Step 7:  Clear Status Register and return to Read Array mode */
   FlashWrite( ANY_ADDR, 0x0050L );   /* Clear Status Register */
   /* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */
   FlashWrite( ANY_ADDR, 0x00FF );    /* Read Array Command */
```

```
   /* Step 8: 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 or FLASH_WRONG_TYPE are 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_BLOCK_FAILED_ERASE (-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 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)       successful operation
   FLASH_PROGRAM_FAIL         (-6)       failure not covered below
   FLASH_OFFSET_OUT_OF_RANGE  (-7)       program range outside device
   FLASH_WRONG_TYPE           (-8)       wrong flash fitted
   FLASH_PROTECTED            (-11)      block to program is protected
   FLASH_VPP_INVALID          (-13)      Vpp is not valid
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.
   Note that the function always programs all addresses within the same page by
   issuing a single program command, as is required by the device architecture.
   Once the program command has completed the function checks the Status
   Register for errors. Any errors are returned without any further attempts to
   program other addresses of the device. The function returns FLASH_SUCCESS
   when all addresses have successfully been programmed.

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:  Determine limits of current set of 4 pages
   Step 5:  Program within the next set of 4 pages
   Step 6:  Wait until the Program/Erase Controller is ready
   Step 7:  Check for any errors
   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 long ulEndSet;       /* Holds the end of the current set of pages */
   unsigned int uStatus;         /* Holds the Status Register reads */
   unsigned int uNumWords;       /* Number of words to be programmmed */
   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: Determine limits of current set of 4 pages */
      ulEndSet = 16L * (ulOff / 16L) + 15L;
      if( ulEndSet > ulLastOff )
         ulEndSet = ulLastOff;
      uNumWords = ulEndSet - ulOff + 1; /* Number of words to be programmed */

      /* Step 5: Program within the next set of 4 pages */
      FlashWrite( ANY_ADDR, 0x0050L );         /* Clear Status Register */
      /* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */
      FlashWrite( ulOff, 0x00E8 );             /* Program Set-up */
      FlashWrite( ulOff, uNumWords - 1 );      /* Number of words */
      while( ulOff <= ulEndSet )
         FlashWrite( ulOff++, *(uArrayPointer++) );   /* Program value */
      FlashWrite( ANY_ADDR, 0x00D0 );          /* Confirm program */

      /* Step 6: Wait until Program/Erase Controller is ready */
      /* TimeOut! */
      do
         uStatus = FlashRead(ANY_ADDR);
      while( (uStatus & 0x0080) == 0x0000 );

      /* Step 7: Check for any errors */
      if( uStatus & 0x0008 )
         iRetVal = FLASH_VPP_INVALID;
      else if( uStatus & 0x0002 )
         iRetVal = FLASH_PROTECTED;
      else if( uStatus & 0x0020 )
         iRetVal = FLASH_PROGRAM_FAIL;
      else if( uStatus & 0x0010 )
         iRetVal = FLASH_PROGRAM_FAIL;
   }

   /* Step 8: Clear Status Register and return to Read Array mode */
   FlashWrite( ANY_ADDR, 0x0050 ); /* Clear Status Register */
   /* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */
   FlashWrite( ANY_ADDR, 0x00FF ); /* Read Array Command */

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

/*****************************************************************************
Function:     int FlashBlockProtect( unsigned char ucBlock )
Arguments:    ucBlock holds the block number to protect
Return Value: The function returns the following conditions:
   FLASH_SUCCESS        (-1)
   FLASH_BLOCK_INVALID  (-5)
   FLASH_WRONG_TYPE     (-8)
   FLASH_PROTECTED      (-11)
```

```
   FLASH_VPP_INVALID    (-13)
   FLASH_PROTECT_FAIL   (-18)
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 if the block really needs protecting as it may already be
            protected
   Step 4:  Protect the block
   Step 5:  Wait until Program/Erase Controller is ready
   Step 6:  Check for any errors
   Step 7:  Clear Status Register and return to Read Array mode
   Step 8:  Read the block ucBlock to check protection status
*****************************************************************************/
int FlashBlockProtect( unsigned char ucBlock )
{
   int iRetVal; /* Store result */
   unsigned int uStatus;

   /* 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 to see if the block number ucBlock is valid */
   iRetVal = FlashAutoSelect( (int)ucBlock );
   if (iRetVal == FLASH_BLOCK_INVALID)
      return FLASH_BLOCK_INVALID;

   /* Step 3: Check if the block really needs protecting as it may already
              be protected */
   else if( iRetVal == FLASH_BLOCK_PROTECTED )
      return FLASH_PROTECTED;

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

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

   /* Step 6: Check for any errors */
   if( uStatus & 0x0008 )
      iRetVal = FLASH_VPP_INVALID;
   else if( uStatus & 0x0020 )
      iRetVal = FLASH_PROTECT_FAIL;
   else if( uStatus & 0x0002 )
      iRetVal = FLASH_PROTECT_FAIL;
   else if( uStatus & 0x0010 )
      iRetVal = FLASH_PROTECT_FAIL;
   else
      iRetVal = FLASH_SUCCESS;
```

```
   /* Step 7: 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 8: Read the block ucBlock to check protection status */
   if(iRetVal != FLASH_SUCCESS)
      return iRetVal;
   else if (FlashAutoSelect( (int)ucBlock ) != FLASH_BLOCK_PROTECTED)
      return FLASH_PROTECT_FAIL;
   else
      return FLASH_SUCCESS;
}

/*****************************************************************************
Function:     int FlashChipUnprotect( void )
Arguments:    none
Return Value: The function returns the following conditions:
   FLASH_SUCCESS        (-1)
   FLASH_BLOCK_INVALID  (-5)
   FLASH_WRONG_TYPE     (-8)
   FLASH_UNPROTECT_FAIL (-16)
   FLASH_UNPROTECTED    (-10)
   FLASH_VPP_INVALID    (-13)
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:  Unprotect the flash memory
   Step 3:  Wait until Program/Erase Controller is ready
   Step 4:  Check for any errors
   Step 5:  Clear Status Register and return to Read Array mode
*****************************************************************************/
int FlashUnprotect( void )
{
   int iRetVal; /* Store result */
   unsigned int uStatus;

   /* 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: Unprotect the flash memory */
   FlashWrite( ANY_ADDR, 0x0060 );
   FlashWrite( ANY_ADDR, 0x00D0 );

   /* Step 3: Wait until Program/Erase Controller is ready */
   do
      uStatus = FlashRead(ANY_ADDR);
      /* TimeOut! */
   while( (uStatus&0x0080) == 0x0000 );

   /* Step 4: Check for any errors */
   if( uStatus & 0x0008 )
      iRetVal = FLASH_VPP_INVALID;
   else if( uStatus & 0x0020 )
```

```
      iRetVal = FLASH_UNPROTECT_FAIL;
   else if( uStatus & 0x0002 )
      iRetVal = FLASH_UNPROTECT_FAIL;
   else if( uStatus & 0x0010 )
      iRetVal = FLASH_UNPROTECT_FAIL;
   else
      iRetVal = FLASH_SUCCESS;

   /* Step 5: 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 */

   return iRetVal;
}

/*****************************************************************************
Function:     int FlashSetBurstConfig( unsigned long ulBurstConfig )
Arguments:    ulBurstConfig specifies the Burst Configuration Register setting.
   The user should only set this value by bit-wise ORing of the bit mask
   definitions provided in the header file "c1270_16.h".
Return Value: The function returns the following conditions:
   FLASH_SUCCESS        (-1)
   FLASH_WRONG_TYPE     (-8)
   FLASH_CONFIG_INVALID (-23)
Description:  This function changes the configuration of read accesses to the
   Flash memory. The function first checks that the configuration specified by
   argument ulBurstConfig is valid for the Flash fitted. Then it issues the
   command required to change the Burst Configuration of the device.

Notes:        After the Burst Configuration Register is modified by this
   function, read accesses from the device will only be successful if the bus
   configuration of the microprocessor is set up accordingly.
              The Y-latency bit of the Burst Configuration Register has a
   different meaning depending on the X-latency. So there is an extra check to
   adjust the Y-latency bit of the Burst Configuration Register based on the
   value specified by the argument ulBurstConfig.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check that the specified Burst Configuration is valid
   Step 3:   Adjust the Burst Configuration Register Y-latency
   Step 4:  Issue the Set Burst Configuration Register Command
*****************************************************************************/
int FlashSetBurstConfig( unsigned long ulBurstConfig )
{
   unsigned long ulBLen;  /* Burst length */
   unsigned long ulXLat;  /* Burst X-latency */

   /* 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 specified Burst Configuration is valid */
   if( ulBurstConfig & FLASH_BCR_MODE_MASK == FLASH_BCR_ASYNCHRONOUS )
      ; /* No configuration checks needed for asynchronous mode */
   else /* Synchronous mode: check the burst length, X-latency and Y-latency */
   {
```

```
      /* Check that the burst length setting is valid */
      ulBLen = ulBurstConfig & FLASH_BCR_LENGTH_MASK;
      if( (ulBLen != FLASH_BCR_LENGTH_1) && (ulBLen != FLASH_BCR_LENGTH_2) &&
          (ulBLen != FLASH_BCR_LENGTH_4) && (ulBLen != FLASH_BCR_LENGTH_8) &&
          (ulBLen != FLASH_BCR_CONTINUOUS) )
         return FLASH_CONFIG_INVALID;
      /* Check that the X-latency setting is valid */
      ulXLat = ulBurstConfig & FLASH_BCR_XLAT_MASK;
      switch( ulXLat )
      {
      case FLASH_BCR_XLAT_8:
      case FLASH_BCR_XLAT_9:
      case FLASH_BCR_XLAT_12:
         /* No other checks needed for these values of X-latency */
         break;
      case FLASH_BCR_XLAT_7:
      case FLASH_BCR_XLAT_10:
      case FLASH_BCR_XLAT_11:
         /* Need to make sure that the burst length is continuous */
         if( ulBLen != FLASH_BCR_CONTINUOUS )
            return FLASH_CONFIG_INVALID;
         break;
      case FLASH_BCR_XLAT_13:
      case FLASH_BCR_XLAT_15:
         /* Need to make sure that the Y-latency setting is valid */
         if( ulBurstConfig & FLASH_BCR_YLAT_MASK == FLASH_BCR_YLAT_1 )
            return FLASH_CONFIG_INVALID;
         else
            /* Step 3: Adjust the Burst Configuration Register Y-latency */
            ulBurstConfig = ulBurstConfig & ~FLASH_BCR_YLAT_MASK;
         break;
      default:
         return FLASH_CONFIG_INVALID;
      }
   }

   /* Step 4: Issue the Set Burst Configuration Register Command */
   FlashWrite( ulBurstConfig << 1, 0x0060 );   /* First cycle */
   FlashWrite( ulBurstConfig << 1, 0x0003 );   /* Second cycle */

   return FLASH_SUCCESS;
}

/*****************************************************************************
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 Overlay Command Failed",
                        "Flash Overlay Disabled already",
                        "Flash Overlay Enabled already",
                        "Flash CFI Failure",
                        "Flash Configuration Invalid"
                       };

   /* 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 be faulty.
*******************************************************************************

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:     Check that Vpp is not floating. Try erasing the block again. If
   this fails, the device may be faulty and need replacing.
*******************************************************************************
```

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

Error Name:   FLASH_VPP_INVALID
Notes:        Applies to M28 (and M58) 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
Notes:        Applies to M29 series Flash only. This error condition should not
   occur when using this library.
Return Value: -15
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 the chip unprotect command
   was unsuccessful.
Solution:     Check that Vpp is not floating but is tied to a valid voltage. Try
   the command again. If it fails a second time then it is likely that the
   device cannot be unprotected and will need to be replaced.
*******************************************************************************

Error Name:   Flash_BANK_INVALID
Notes:        This applies to M59DRxxx series Flash only. This error condition
   should not occur when using this library.
Return Value: -17
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:     Check that Vpp is not floating but is tied to a valid voltage. Try
   the command again. If it fails a second time then the block cannot be
   protected and it may be necessary to replace the device.
*******************************************************************************

Error Name:   Flash_OVERLAY_FAIL
Notes:        This applies only to Flash memories with Overlay Blocks. This
   error condition should not occur when using this library.
Return Value: -19
Description:  This error indicates that any Overlay Block associated function
   has failed.

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

Error Name:   Flash_OVERLAY_DISABLED
Notes:        This applies only to Flash memories with Overlay Blocks. This
   error condition should not occur when using this library.
Return Value: -20
Description:  This error informs the user that a OverlayBlockDisable command
   was not neccesary because the Overlay Block was already disabled.

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

Error Name:   Flash_OVERLAY_ENABLED
Notes:        This applies only to Flash memories with Overlay Blocks. This
   error condition should not occur when using this library.
Return Value: -21
Description:  This error informs the user that a OverlayBlockEnable command
   was not necessary because the Overlay Block was already enabled.

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

Error Name:   Flash_CFI_FAIL
Return Value: -22
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_CONFIG_INVALID
```

```
Return Value: -23
Description:  This error return value indicates that the specified configuration
   is not valid or not supported for the device being used.
Solution:     Check the specified configuration. Identify a configuration
   supported by the device and call the function again with this new setting.

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

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

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

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

littlediode
COMPONENTS FOR ALL

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

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

Feel free to visit us today at our online store:

LittleDiode.com

Looking forward to providing you with the best possible service.