SGS-THOMSON MICROELECTRONICS

AN942 APPLICATION NOTE

SOFTWARE DRIVERS for M29F200 FLASH MEMORY

CONTENTS

- Introduction
- The M29F200 Programming Model
- Modifying code from Am29F200
- Generating SGS-THOMSON Code
- C Library Functions
- Adapting the Software for the Target System
- Limitations of the code
- Connection to Common Microprocessors
- Conclusion
- Source Code
  - M29F200A.H Header file for 16 bit C Routines library
  - M29F200A.C 16 bit C Routines library
  - M29F200B.H Header file for 8 bit C Routines library
  - M29F200B.C 8 bit C Routines library

These files may be downloaded from www.st.com or obtained from any Sales offices on PC compatible floppy disk.

INTRODUCTION

This application note provides library source code in C for the M29F200 Flash memory. There are two different M29F200 parts, the M29F200B and M29F200T. In this application note the term M29F200 can refer to either part.

The application note includes listings of the source code which is also available in file form. The m29f200a files contain libraries for accessing the M29F200 Flash memories in 16-bit bus mode, whereas the m29f200b files contain the 8-bit drivers.

An overview of the programming model for the M29F200 is given. The programming differences between the M29F200 and AMD's Am29F200 are described. Advice on how to modify programs already written for AMD's device to work with the ST device is included.

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.

Brief hardware connections to some common microprocessors are provided at the end of the application note to help the designer understand the bus requirements of the M29F200.

This application note does not replace the M29F200 Data Sheet. It refers to the Data Sheet throughout and it is necessary to have a copy in order to follow some of the explanations.

The software and accompanying documentation has been fully tested on a target platform. It is small in size and can be applied to any target hardware.

THE M29F200 PROGRAMMING MODEL

The M29F200 is a 256K x 8 or 128K x 16 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 7 blocks of varying sizes. Each block can be erased individually, or the whole chip can be erased at once, erasing all 2 Mb.

The M29F200 is a single voltage device. It differs from first generation devices which require a 12V supply to program or erase. The M29F200 is therefore easier to use since the hardware does not need to cater for special bus signal levels. The voltages needed to erase the device are generated by charge pumps inside the device.

Included in the device is a Program/Erase Controller (P/E.C.). With first generation flash memory devices the software had to manually program all of the bytes to 00h before erasing to FFh using special programming sequences. The P/E.C. in the M29F200 allows a simpler programming model to be used. The P/E.C. takes care of all the necessary steps required to erase and program the memory. This has led to improved reliability so that in excess of 100,000 program/erase cycles are guaranteed per block on the device.

The M29F200 does, however, require some high voltage bus signals if all of the functionality of the device is to be accessed. Each block can be protected against accidental programming or erasure. Protecting and unprotecting the blocks requires V_{ID} (about 11V) on some of the pins.

Most applications of the device will not include these functions.

However, blocks may be preprogrammed, protected and unprotected by an EPROM programmer prior to fitting into the hardware. Unprotected blocks may still be used to store data and parameters. By protecting a block, accidental data loss through software failure cannot occur.

MODES

All write accesses to the M29F200 go to the P/E.C. which decodes them as commands. These commands are used to put the M29F200 into various modes, which are:

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

– The Read Array mode is the reset state of the M29F200. In this mode the device behaves as a ROM. A read cycle outputs the data stored at the specified address on the data bus.
– The Auto Select mode allows the user to read the Electronic Signature and Block Protection Status of the device. The electronic signatures (manufacturer and device codes) or the block protection status are accessed by reading different addresses whilst in the Auto Select mode.
– During the Program/Erase mode of the M29F200 a read cycle will output the Status Register of the P/E.C. The Status Register contains valuable information about the program or erase operation which is happening or has finished.
– During an erase cycle the M29F200 can be temporarily placed in Erase Suspend mode. In this mode the blocks not being erased may be read as if in the Read Array mode. This allows the user to access information stored in the device immediately rather than waiting until the erase completes, typically 1.0s for block erases.

The Instructions Table of the M29F200 Data Sheet describes the sequence of Command Bytes and the respective addresses which need to be written to the P/E.C. to change mode. Note that the instructions depend on whether the device is accessed in 8 bit or 16 bit mode.

To change between modes write accesses to the specified addresses with the correct data are required. For example, using 16 bit mode entering the program/erase mode and programming 9465h to the address 03E2h requires the user to write the following sequence (in C):

```
*(unsigned int*)(0x5555) = 0x00AA;
*(unsigned int*)(0x2AAA) = 0x0055;
*(unsigned int*)(0x5555) = 0x00A0;
*(unsigned int*)(0x03E2) = 0x9465;
```

In 8 bit mode writing 65h to address 07C4h would be:

```
*(unsigned char*)(0xAAAA) = 0xAA;
*(unsigned char*)(0x5555) = 0x55;
*(unsigned char*)(0xAAAA) = 0xA0;
*(unsigned char*)(0x07C4) = 0x65;
```

Note that, due to the organisation of the address bus and data bus, this example writes 65h to the same address for both 16 bit and 8 bit modes. The internal organisation of the M29F200 is such that the LSB of the 16 bit mode is stored at the lower address of the 8 bit mode.

This example assumes that the M29F200 address 0000h 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.

The program/erase mode is the most complex since it allows the user to verify that the programming or erasing has completed and has been performed correctly. During the program or erase cycle any read from the P/E.C. will read from the Status Register. The Status Register bits are described in the Status Register Bits table of the M29F200 Data Sheet.

During the program or erase cycle the user can verify that the erase is progressing by following the Data Polling Flowchart figure of the M29F200 Data Sheet. Alternatively the Data Toggle Flowchart figure can be used. Note that the Data Polling Flowchart is easier to implement. This technique has been used exclusively in the library routines described in this application note.

The end of a program cycle can be identified using the Data Polling Flowchart when DQ7 of the Status Register is the same as bit 7 of the byte being programmed. Erasing writes FFh to the memory, therefore DQ7 will be 0 during the erase cycle and 1 following the erase cycle.

The error bit, DQ5 of the Status Register, can be used to check if an error has occurred. If no error occurs then the Data Polling Flowchart figure of the M29F200 Data Sheet will give a PASS result. When an error occurs DQ7 will continue to hold the complement of bit 7 of the programmed byte and DQ5 will be set. After an error the user will have to issue a Reset (RST) command to enter Read Array mode before continuing.

When programming an error may occur if the address was not previously erased, therefore erasing the block containing the address and trying again will work. Remember, however, that all of the other bytes will need to be reprogrammed. During the program operation it is only possible to change bits from 1 to 0. Attempting to change a 0 to a 1 using the program command will fail.

MODIFYING CODE FROM Am29F200

If the user has already developed code which supports the Am29F200 devices then modifying the drivers to include the M29F200 should be straightforward.

In many cases there will be no changes required to the system software. The M29F200 and the Am29F200 are identical in their operation. The P/E.C. accepts the same commands on each device and the Status Register has the same bit definitions.

The only difference between the devices is in the Manufacturer Code and the Device Code read during the Auto Select mode. Table 1 shows the codes for the devices.

Consider the following example of code for a device in 16 bit mode:

```
unsigned int device_code( void )
{
   /* Write auto select sequence */
   FlashWrite( 0x5555L, 0x00AA );
   FlashWrite( 0x2AAAL, 0x0055 );
   FlashWrite( 0x5555L, 0x0090 );

   /* Read Device Code */
   return FlashRead( 0x0001L );
}
```

Table 1. Manufacturer and Device Codes Table

Device	Manufacturer Code	Device Code
M29F200T	20h	(00)D3h
M29F200B	20h	(00)D4h
Am29F200T	01h	(22)51h
Am29F200B	01h	(22)57h

The function `device_code()` may be used to identify the device fitted. The functions `FlashWrite()` and `FlashRead()` write to and read from the memory. Their use is explained in detail later in this application note.

In order to hide the difference between the M29F200 and the Am29F200 from the user's source code the function could be changed to the following:

```
unsigned int device_code( void )
{
  /* Write auto select sequence */
  FlashWrite( 0x5555L, 0x00AA );
  FlashWrite( 0x2AAAL, 0x0055 );
  FlashWrite( 0x5555L, 0x0090 );

  /* Check for ST */
  if( FlashRead(0x0000L) == 0x0020 )
  {

    if( FlashRead(0x0001L) == 0x00D3 )
      return 0x2251;

    if( FlashRead(0x0001L) == 0x00D4
      return 0x2257;
  }

  /* Read Device Code */
  return FlashRead( 0x0001L );
}
```

A similar condition will need to be put in any functions which read the manufacturer code to ensure that the algorithms identify the device as an AMD part. No other changes to the algorithms will be necessary due to the compatibility of ST and AMD devices.

Referring back to the modified version of the example function `'device code'` above, it will be clear that higher level functions cannot differentiate between the Am29F200T and the M29F200T from the return value of `'device_code'`. For applications, such as flash programmers, which need to distinguish between SGS-THOMSON and AMD parts, the original version of the `'device_code'` function must be retained. In this case, higher level functions which rely on the return value of `'device_code'` must be modified to recognise the SGS-THOMSON parts.

If the user is developing a new software to support the AMD Am29F200B as well as the ST M29F200 or the original AMD Am29F200, he should be aware that AMD, changing revision of their product, have also changed the number of address bits specified in the coded cycle during the write operations.

When writing command sequences to the ST M29F200 or the Am29F200, the address lines A11 to A14 need to be set correctly, while A15 to A16 can be set to any value whereas for the AMD Am29F200B the address lines A11 to A16 can be set to any value.

The source code in this application note will work with any of the parts, including the Am29F200B. However, care should be taken when fitting an M29F200 in place of an Am29F200B: address bits A11 to A14 must be set to the appropriate value during coded cycles. This practically means that the address value must be set as per product datasheet on 4 hexadecimal digits for the M29F200.

GENERATING SGS-THOMSON CODE

Low-level functions (drivers) have been provided to simplify the process of developing application code in C for the SGS-THOMSON M29F200 Flash memories. This enables users to concentrate on writing the high level functions required for their particular applications. These high level funtions 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.

Flash memories are typically used to store source code and data in embedded systems, especially when field updates are likely to be required; a few example applications are personal data organisers, mobile phones or PCMCIA cards.

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

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 the block protection levels of the device.

`FlashBlockErase()` is used to erase one or more blocks in the device. Multiple blocks will be erased simultaneously to reduce the overall erase time. This function checks that none of the blocks specified are protected and does not erase any blocks if some of the specified blocks are protected.

`FlashChipErase()` is used to erase the entire chip. It will not erase any blocks if there is a protected block on the chip.

The functions rely on the user writing short, simple functions to read and write to the flash in their particular target hardware.

`FlashRead()` must be written to read a value from the flash.

`FlashWrite()` must be written to write a value to the flash.

`FlashPause()` must be written to provide a timer with microsecond resolution. This is used to wait while the flash recovers from some conditions.

An example of these functions is provided in the source code.

In many instances these functions can be written as macros and therefore will not incur the function call time overhead. The two functions which perform the basic I/O to the device have been provided for users who have awkward systems. For example where the addressing system is peculiar or the data bus has D0..D7 of the device on D8..D15 of the microprocessor. They allow any user to quickly adapt the code to virtually any target system.

Throughout the functions, assumptions have been made on the data types. These are:

A `char` is 8 bits (1 byte). This is not the case in all microcontrollers. Where it is not it will be necessary to mask the unused bits of the word in the user's `FlashRead()` function.

An `int` is 16 bits (2 bytes). Again, like the `char`, if this is not the case it will be necessary to use a variable type which is 16 bits or longer and mask bits above 16 bits.

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

Two approaches to the addressing are available: the desired address in the memory 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.

ADAPTING THE SOFTWARE FOR THE TARGET SYSTEM

Before using the software in the Target System the user needs to do the following:

1. Define USE_M29F200T or USE_M29F200B depending on whether an M29F200T or M29F200B is fitted. The top of the source file defines USE_M29F200T.
2. Write `FlashRead()`, `FlashWrite()` and `FlashPause()` functions appropriate to the Target Hardware.
3. Search through the code for the `/* DSI */` and `/* ENI */` comments and disable/enable interrupts at the appropriate points.

The example `FlashRead()` and `FlashWrite()` functions provided in the source code should give the user a good idea of what is required and can be used in many instances without much modification.

To test the source code in the Target System start by simply reading from the M29F200. If it is erased then only FFh data should be read. Next read the Manufacturer and Device codes and check they are correct. If these functions work then it is likely that all of the functions will work but they should all be tested thoroughly.

The programmer needs to take extra care when the device is accessed during an interrupt service routine. Three situations exist which must be considered:

1. When the device is in Read Array mode interrupts can freely read from the device.
2. Interrupts which do not access the device may be used during the Program, Autoselect and Chip Erase functions.
3. During the time critical section of the Block Erase function interrupts are not permitted. An interrupt during this time may cause a time-out and result in some of the blocks not being erased correctly.

The programmer should also take care when a Reset is applied during program or erase operations. The flash will be left in an indeterminate state and data could be lost.

C does not provide a standard library function for disabling interrupts. Furthermore different applications have different tolerances on when interrupts may be disabled. Therefore no protection from the mis-use of interrupts could be incorporated into the library source code. This is left to the user.

It is strongly recommended that the user disables interrupts where the `/* DSI */` comments are placed in the source code. If this is not possible then the user should erase one block at a time.

LIMITATIONS OF THE SOFTWARE

The software provided does not implement a full set of the M29F200's functionality. It is left to the user to implement the Erase Suspend, Block Protect and Chip Unprotect commands of the device. The Standby mode is a hardware feature of the device and cannot be controlled through software.

Care should be taken in some of the `for()` 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.

CONNECTION TO COMMON MICROPROCESSORS

The M29F200 can be connected easily to a variety of microprocessors. In the examples given here the connection to the ST10R163 and the MC68330 show how it can be used with a non-multiplexed bus

whereas the i80960SA shows how to connect to a multiplexed bus. The connection to the MC68330 is given as a byte-wide example whereas the others are shown in word-wide mode.

Connection to the ST10R163

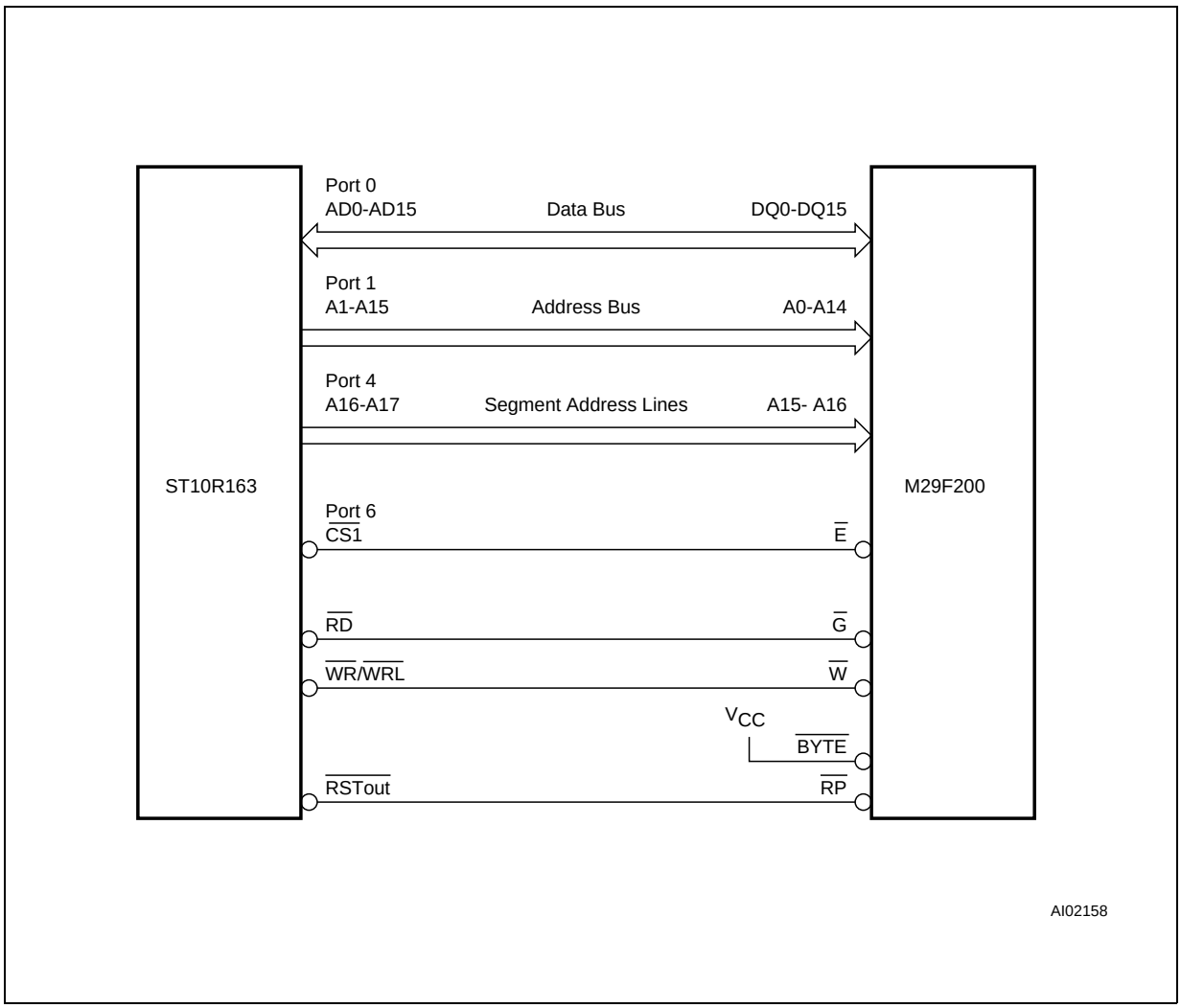
Figure 1 shows an example of how to connect the ST10R163 to the M29F200. In this example the ST10R163 accesses the M29F200 in 16-bit non-multiplexed bus mode and the M29F200 operates in word-wide (x16) mode.

The Port6 $\overline{CS1}$ output of the ST10R163 is used to enable the M29F200. The address range over which $\overline{CS1}$ enables the M29F200 is selected by configuring BUSCON1 of the ST10R163.

The data bus is fully connected between the two devices.

In order to ensure that words are accessed correctly, Port1 A1 of the ST10R163 must be connected to A0 of the M29F200. (Note that software must ensure that A0 of the ST10R163 is Low when words are accessed, or an internal error trap will occur.) Consequently A1-17 of the ST10R163 are connected to A0-16 of the M29F200, giving an address space of 128K Words.

Figure 1. Connection of the M29F200 to the ST10R163



Connection to the Intel 80960SA

Figure 2 shows an example of how to connect a i80960SA to the M29F200. The i80960SA has a full 32-bit address bus. However the address and data buses are multiplexed making it necessary to latch addresses before data access. Furthermore, the i80960SA does not directly provide the control signals required for the M29F200 ($\overline{G}$, $\overline{E}$ and $\overline{W}$); these must be generated by external logic.

The chip enable logic in the example can be made to map the 128K words memory space of the M29F200 to any 128K Word boundary of the 2G Word address space of the i80960SA.

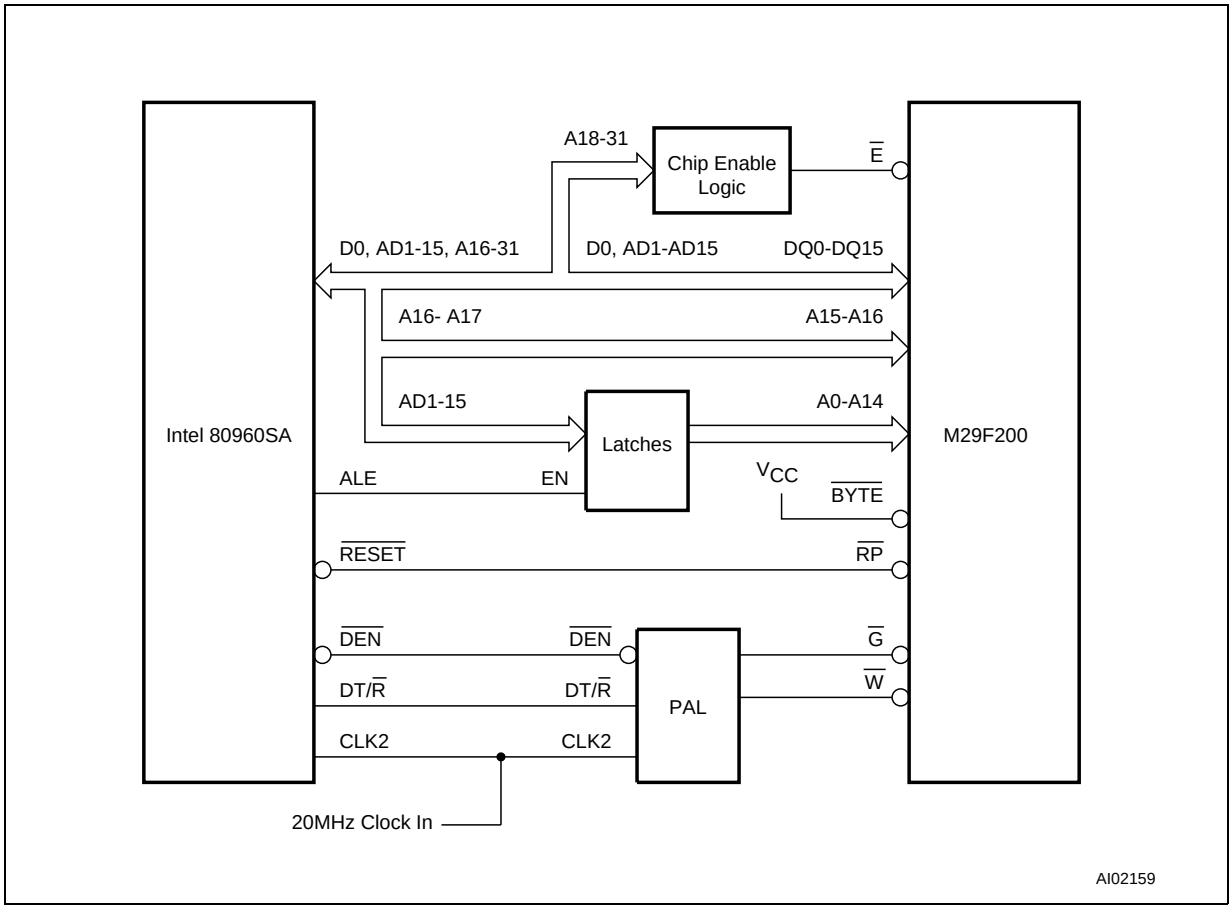
The M29F200 is configured to operate in word-wide (x16) mode by pulling $\overline{BYTE}$ to V_{IH}. The i80960SA only operates in 16-bit access mode. The least significant address bit is A1 and therefore, as with the ST10R163, the AD1 signal of the i80960SA connects to A0 of the M29F200. The other address bits are shifted likewise.

As the i80960SA uses a multiplexed bus, the least significant word of the address bus needs to be latched on the falling edge of ALE. These lines are used for data in the second half of the access cycle during which time the address needs to be held for the M29F200.

The PAL provides the control signals which drive the M29F200. This example provides zero wait state access to an M29F200-70 for a i80960SA at 10MHz.

The i80960SA drives the DT/$\overline{R}$ output Low to indicate a read cycle. During the read cycle the $\overline{G}$ input of the M29F200 should only be asserted when the i80960SA has released the bus. This can be up to 18ns after $\overline{DEN}$ of the i80960SA goes Low. Therefore the $\overline{G}$ needs to be asserted on the rising edge of CLK2

Figure 2. Connection of the M29F200 to the 80960SA



following the falling edge of $\overline{DEN}$. $\overline{G}$ should be kept asserted for two cycles of CLK2: it is set High at the second rising edge of CLK2.

The i80960SA indicates a write cycle by driving DT/$\overline{R}$ High. During the write cycle the rising edge of $\overline{W}$ on the M29F200 controls latching of the data. The $\overline{DEN}$ output of the i80960SA cannot be used directly for this purpose as its rising edge occurs when data is removed from the bus. Instead the $\overline{W}$ input of the M29F200 is asserted on the rising edge of CLK2 following the falling edge of $\overline{DEN}$. Then, in order to latch the data, $\overline{W}$ is set High on the following rising edge of CLK2.

Connection to the MC68330

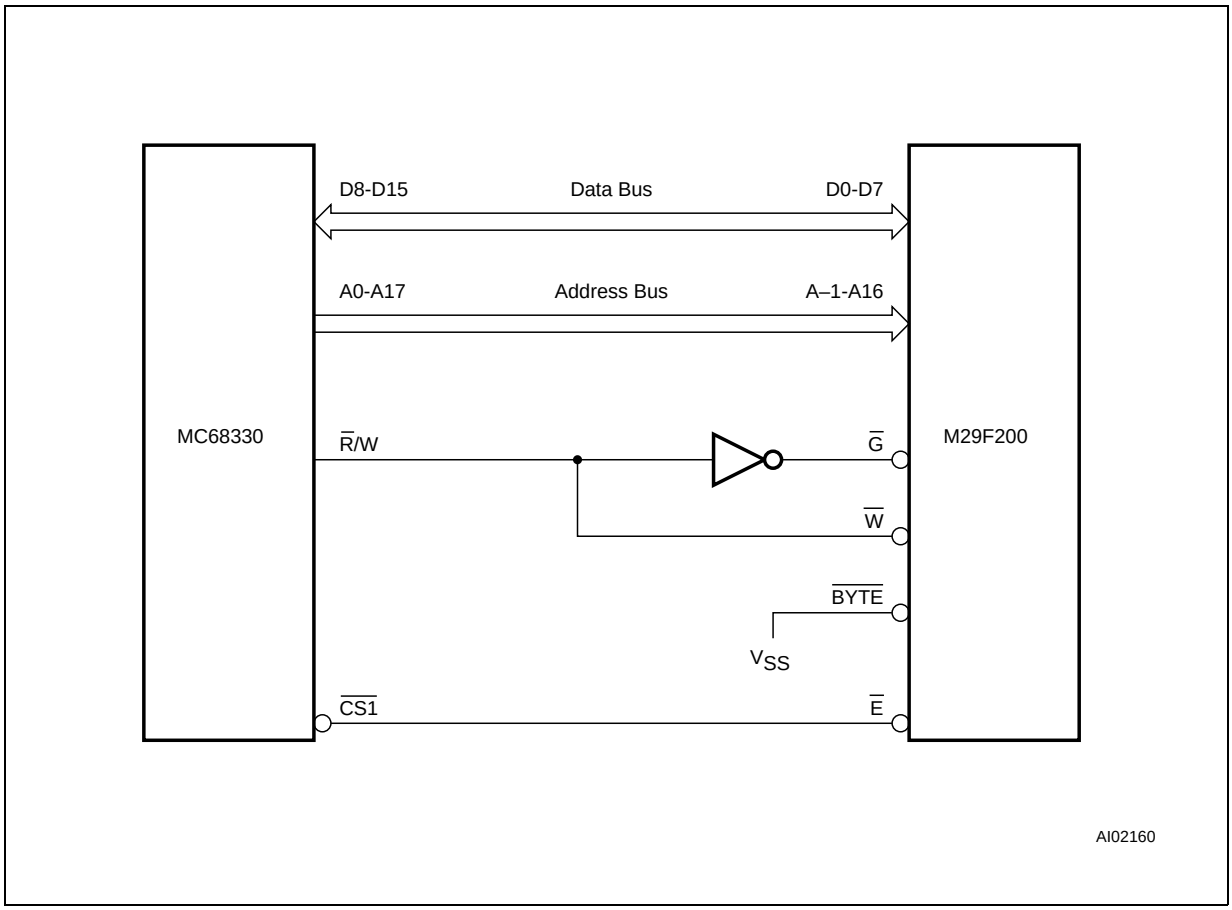
The MC68330 has a non-multiplexed 16-bit data bus; however the example in Figure 3 shows the configuration for byte-wide access purely to illustrate the byte-wide mode of the M29F200.

The Motorola standard for reading bytes on a 16-bit bus is to access the byte on the upper data lines, hence D0 of the M29F200 is connected to D8 of the MC68330, with the other data lines following.

The MC68330 has four chip selects, one of which can readily be used to select the M29F200 device. Further more during a write cycle the data is held on the data bus for 15ns following the rising edge of $\overline{CS1}$, which can thus be used to latch the data into the M29F200.

The MC68330 provides one output to indicate either a read or a write. An inverter is required to transform this for the output enable $\overline{G}$ pin of the M29F200. During the read cycle the MC68330 does not require the data to be held after $\overline{CS1}$ goes high, allowing the M29F200 $\overline{E}$ input to be driven directly from $\overline{CS1}$.

Figure 3. Connection of the M29F200 to the MC68330



CONCLUSION

The M29F200 single voltage Flash memory is an ideal product for embedded and other computer systems, able to be easily interfaced to the microprocessor and driven with simple software drivers written in the C language.

Full product information can be found at *www.st.com*, queries may be sent by email to *ask.memory@st.com*.

```
/****M29F200A.H*Header File for M29F200A.C************************************

   Filename:    m29f200a.h
   Description: Header file for m29f200a.c. Consult the C file for details

   Copyright (c) 1997 SGS-THOMSON Microelectronics.

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

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

/******************************************************************************
Error Conditions and return values.
See end of C file for explanations and help
******************************************************************************/
#define FLASH_BLOCK_PROTECTED      (0x01)
#define FLASH_BLOCK_UNPROTECTED    (0x00)
#define FLASH_BLOCK_NOT_ERASED     (0xFF)
#define FLASH_BLOCK_ERASE_FAILURE  (0xFE)
#define FLASH_BLOCK_ERASED         (0xFD)
#define FLASH_SUCCESS              (-1)
#define FLASH_POLL_FAIL            (-2)
#define FLASH_TOO_MANY_BLOCKS      (-3)
#define FLASH_MPU_TOO_SLOW         (-4)
#define FLASH_BLOCK_INVALID        (-5)
#define FLASH_PROGRAM_FAIL         (-6)
#define FLASH_OFFSET_OUT_OF_RANGE  (-7)
#define FLASH_WRONG_TYPE           (-8)
#define FLASH_BLOCK_FAILED_ERASE   (-9)


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

```
/****M29F200A.C*2Mb Flash Memory*16bit mode************************************

   Filename:    m29f200a.c
   Description: Library routines for the M29F200 2Mbit (128kx16) Flash Memory.
                16 bit drivers.

   Version:     1.01
   Date:        27/8/97
   Author:      Brendan Watts, OTS & Alexandre Nairac
   Copyright (c) 1997 SGS-THOMSON Microelectronics.

   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   22/07/97  Initial Release of the Software.
   1.01   27/08/97  Minor revisions to the Software.

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

   This source file provides library C code for using the M29F200 devices (both
   M29F200T and M29F200B) in 16 bit mode. A separate file is available for users
   who wish to access the device in 8 bit mode.

   The following functions are available in this library:

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

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

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

      FlashWrite() for writing a word to the flash
```

(Cont'd in the next page)

```
      FlashRead()  for reading a word from the flash
      FlashPause() for timing short pauses (in micro seconds)

   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.

   C does not include a method for disabling interrupts to keep time-critical
   sections of code from being interrupted. The user may wish to disable
   interrupts during parts of the code to avoid the FLASH_MPU_TO_SLOW error
   from occuring if an interrupt occurs at the wrong time. Where interrupts
   should be disabled and re-enabled there is a /# DSI! #/ or /# ENI! #/
   comment.

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

#include <stdlib.h>

#include "m29f200a.h"        /* Header file with global prototypes */

#define USE_M29F200T


/*****************************************************************************
Constants
*****************************************************************************/
#define COUNTS_PER_MICROSECOND (200)
#define MANUFACTURER_ST (0x0020)
#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 */


#ifdef USE_M29F200T
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0 */
   0x08000L,  /* Start offset of block 1 */
   0x10000L,  /* Start offset of block 2 */
   0x18000L,  /* Start offset of block 3 */
   0x1C000L,  /* Start offset of block 4 */
   0x1D000L,  /* Start offset of block 5 */
   0x1E000L   /* Start offset of block 6 */
};
```

(Cont'd in the next page)

```
#define EXPECTED_DEVICE (0x00D3) /* M29F200T */
#endif

#ifdef USE_M29F200B
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0 */
   0x02000L,  /* Start offset of block 1 */
   0x03000L,  /* Start offset of block 2 */
   0x04000L,  /* Start offset of block 3 */
   0x08000L,  /* Start offset of block 4 */
   0x10000L,  /* Start offset of block 5 */
   0x18000L   /* Start offset of block 6 */
};

#define EXPECTED_DEVICE (0x00D4) /* M29F200B */

#endif

#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))
#define FLASH_SIZE (0x20000L) /* 128k x16 */



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

The following functions are only needed in this module.
*****************************************************************************/

static unsigned int FlashWrite( unsigned long ulOff, unsigned int uVal );
static void FlashPause( unsigned int uMicroSeconds );
static int FlashDataPoll( unsigned long ulOff, unsigned int uVal );
static int FlashBlockFailedErase( unsigned char ucBlock );


/*****************************************************************************
Function:    unsigned int FlashWrite( unsigned long ulOff, unsigned int uVal )
Arguments:   ulOff is word offset into the flash to write to.
   uVal is the value to be written
Returns:     uVal
Description: This function is used to write a value 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.
```

(Cont'd in the next page)

```
   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 word offset into the flash to read from.
Returns:     The unsigned int at the word offset
Description: This function is used to read a word from the flash. On many
   microprocessor systems a macro can be used instead, increasing the speed of
   the flash routines. For example:

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

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

Pseudo Code:
  Step 1: Return the value at word offset ulOff

*******************************************************************************/
unsigned int FlashRead( unsigned long ulOff )
{
   /* Step 1 Return the value at word offset ulOff */
   return BASE_ADDR[ulOff];
}




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

(Cont'd in the next page)

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

Pseudo Code:
   Step 1: Compute count size for pause.
   Step 2: Count to the required size.

*************************************************************/
static void FlashPause( unsigned int uMicroSeconds )
{
   volatile unsigned long ulCountSize;

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

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



/*****************************************************************************
Function:      void FlashReadReset( void )
Arguments:     none
Return Value:  none
Description:   This function places the flash in the Read 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.

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

Pseudo Code:
   Step 1: write command sequence (see Instructions Table of the Data Sheet)
   Step 2: wait 10us

*****************************************************************************/
void FlashReadReset( void )
{
   /* Step 1: write command sequence */
   FlashWrite( 0x5555L, 0x00AA );  /* 1st Cycle */
   FlashWrite( 0x2AAAL, 0x0055 );  /* 2nd Cycle */
```

(Cont'd in the next page)

```
   FlashWrite( 0x5555L, 0x00F0 );  /* 3rd Cycle */

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

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

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

   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:
      M29F200T   00D3h
      M29F200B   00D4h

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

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

Pseudo Code:
   Step 1:  Send the Auto Select Instruction to the device
   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 */
   FlashWrite( 0x2AAAL, 0x0055 );  /* 2nd Cycle */
   FlashWrite( 0x5555L, 0x0090 );  /* 3rd Cycle */

   /* Step 2: Read the required function */
   if( iFunc == FLASH_READ_MANUFACTURER )
      iRetVal = FlashRead( 0x0000L ); /* A0 = A1 = A6 = 0 */

   else if( iFunc == FLASH_READ_DEVICE_CODE )
```

(Cont'd in the next page)

```
      iRetVal = FlashRead( 0x0001L ); /* A0 = 1, A1 = A6 = 0 */

   else if( (iFunc >= 0) && (iFunc < NUM_BLOCKS) )
      iRetVal = FlashRead( BlockOffset[iFunc] + 0x0002L );
                                              /* A0 = A6 = 0, A1 = 1 */
   else
      iRetVal = FLASH_BLOCK_INVALID;

   /* Step 3: Return to Read Array mode */
   FlashWrite( 0x0000L, 0x00F0 ); /* Use single instruction cycle method */

   return iRetVal;
}
```

```
/*****************************************************************************
Function:     int FlashBlockErase( unsigned char ucNumBlocks,
   unsigned char ucBlock[]  )
Arguments:    ucNumBlocks holds the number of blocks in the array ucBlock
   ucBlock is an array containing the blocks to be erased.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS              (-1)
   FLASH_POLL_FAIL            (-2)
   FLASH_TOO_MANY_BLOCKS      (-3)
   FLASH_MPU_TOO_SLOW         (-4)
   FLASH_WRONG_TYPE           (-8)
   Number of the first protected or invalid block

      The user's array, ucBlock[] is used to report errors on the specified
   blocks. If a time-out occurs because the MPU is too slow then the blocks
   in ucBlocks which are not erased are overwritten with FLASH_BLOCK_NOT_ERASED
   (FFh) and the function returns FLASH_MPU_TOO_SLOW.
      If an error occurs during the erasing of the blocks the blocks in
   ucBlocks which have failed the erase are set to FLASH_BLOCK_ERASE_FAILURE
   (FEh) and the function returns FLASH_POLL_FAIL.
      If both errors occur then the function will set the ucBlock array for
   each type of error (i.e. either to FLASH_BLOCK_NOT_ERASED or to
   FLASH_BLOCK_ERASE_FAILURE). It will return FLASH_POLL_FAIL even though
   the FLASH_MPU_TOO_SLOW has also occured.

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

(Cont'd in the next page)

```
  During the Erase Cycle the Data Polling Flowchart of the Data Sheet is
  followed. The toggle bit, DQ6, is not used. For an erase cycle the data on
DQ7
  will be '0' during the erase and '1' on completion.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check for protected or invalid blocks
   Step 3:  Write Block Erase command
   Step 4:  Check for time-out blocks
   Step 5:  Wait for the timer bit to be set
   Step 6:  Perform Data Polling until P/E.C. has completed
   Step 7:  Return to Read Array mode
*****************************************************************************/
int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[] )
{
   unsigned char ucCurBlock;    /* Range Variable to track current block */
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
   unsigned int FirstRead, SecondRead; /* used to check toggle bit DQ2 */

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

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

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

   /* Step 3: Write Block Erase command */
   FlashWrite( 0x5555L, 0x00AA );
   FlashWrite( 0x2AAAL, 0x0055 );
   FlashWrite( 0x5555L, 0x0080 );
   FlashWrite( 0x5555L, 0x00AA );
   FlashWrite( 0x2AAAL, 0x0055 );
   /* DSI!: Time critical section. Additional blocks must be added every 80us */
   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      FlashWrite( BlockOffset[ucBlock[ucCurBlock]], 0x0030 );

      /* Check for Erase Timeout Period */
      if( (FlashRead( BlockOffset[ucBlock[0]] ) & 0x0008 ) == 0x0008 )
```

(Cont'd in the next page)

```
      break; /* Cannot set any more sectors due to timeout */
}
/* ENI! */

/* Step 4: Check for time-out blocks */
/* if timeout occured then check if last block is erasing or not */
/* Use DQ2 of status register, toggle implies block is erasing */
if( ucCurBlock < ucNumBlocks )
{
   iRetVal = FLASH_MPU_TOO_SLOW;

   FirstRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x0004;
   SecondRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x0004;
   if( FirstRead != SecondRead )
   {
      ucCurBlock++; /* Point to the next block */
   }

   /* Now specify all other blocks as not being erased */
   while( ucCurBlock < ucNumBlocks )
   {
      ucBlock[ucCurBlock++] = FLASH_BLOCK_NOT_ERASED;
   }
}

/* Step 5: Wait for the timer bit to be set */
while( 1 )  /* TimeOut!: If, for some reason, the hardware fails then this
            loop may not exit. Use a timer function to implement a timeout
            from the loop. */
{
   if( ( FlashRead( BlockOffset[ucBlock[0]] ) & 0x0008 ) == 0x0008 )
      break; /* Break when device starts the erase cycle */
}

/* Step 6: Perform data polling until P/E.C. completes, check for errors */
if( FlashDataPoll( BlockOffset[ucBlock[0]], 0xFFFF ) != FLASH_SUCCESS )
{
   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      if( ucBlock[ucCurBlock] == FLASH_BLOCK_NOT_ERASED )
        break; /* The rest of the blocks have not been erased anyway */

      if( FlashBlockFailedErase( ucBlock[ucCurBlock] )
      == FLASH_BLOCK_FAILED_ERASE )
      {
         ucBlock[ucCurBlock] = FLASH_BLOCK_ERASE_FAILURE;
      }
   }
```

(Cont'd in the next page)

```
      iRetVal = FLASH_POLL_FAIL;
   }

   /* Step 7: Return to Read Array mode */
   FlashWrite( 0x0000L, 0x00F0 ); /* Use single instruction cycle method */

   return iRetVal;
}
```

```
/*****************************************************************************
Function:     int FlashChipErase( unsigned char *Results )
Arguments:    Results is a pointer to an array where the results will be stored.
   If Results == NULL then no results are stored.
   Otherwise the results are written to the array if an error occurs. The
   array is left unchanged if the function returns FLASH_SUCCESS.
   The errors written to the array are:
      FLASH_BLOCK_ERASED (FDh)        if the block erased correctly
      FLASH_BLOCK_ERASE_FAILURE (FEh) if the block failed to erased
Return Value: On success the function returns FLASH_SUCCESS (-1)
   If a block is protected then the function returns the number of the block.
   If the erase algorithms fails then the function returns FLASH_POLL_FAIL (-2)
   If the wrong type of flash is accessed then the function returns
   FLASH_WRONG_TYPE (-8)
Description: The function can be used to erase the whole flash chip so long as
   no sectors are protected. If any sectors are protected then nothing is
   erased.
Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: Check that all sectors are unprotected
   Step 3: Send Chip Erase Command
   Step 4: Perform data polling until P/E.C. has completed.
   Step 5: Check for blocks erased correctly
   Step 6: Return to Read Array mode
*****************************************************************************/
int FlashChipErase( unsigned char *Results )
{
   unsigned char ucCurBlock; /* Used to track the current block in a range */
   int iRetVal;              /* Holds the return value */

   /* 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;
```

(Cont'd in the next page)

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

   /* Step 3: Send Chip Erase Command */
   FlashWrite( 0x5555, 0x00AA );
   FlashWrite( 0x2AAA, 0x0055 );
   FlashWrite( 0x5555, 0x0080 );
   FlashWrite( 0x5555, 0x00AA );
   FlashWrite( 0x2AAA, 0x0055 );
   FlashWrite( 0x5555, 0x0010 );

   /* Step 4: Perform data polling until P/E.C. completed */
   iRetVal = FlashDataPoll( 0x0000, 0xFFFF );/* Erasing writes 0xFFFF to flash*/

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

   /* Step 6: Return to Read Array mode */
   FlashWrite( 0x0000, 0x00F0 ); /* Use single instruction cycle method */

   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: On success the function returns FLASH_SUCCESS (-1).
   On failure the function returns FLASH_PROGRAM_FAIL (-6).
   If the address exceeds the address range of the Flash Device the function
```

(Cont'd in the next page)

```
   returns FLASH_ADDRESS_OUT_OF_RANGE (-7) and nothing is programmed.
   If the wrong type of flash is accessed then the function returns
   FLASH_WRONG_TYPE (-8).
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 is not erased first.

Pseudo Code:
   Step 1: Check that the flash is of the correct type
   Step 2: Check the offset range is valid.
   Step 3: While there is more to be programmed
   Step 4:  Program the next byte
   Step 5:  Perform data polling until P/E.C. has completed.
   Step 6:  Update pointers
   Step 7: End of While Loop
   Step 8: Return to Read Array mode
*****************************************************************************/
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 */

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

   /* Step 2: Check the address 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 )
   {
      /* Step 4: Program the next byte */
      FlashWrite( 0x5555L, 0x00AA );  /* 1st cycle */
      FlashWrite( 0x2AAAL, 0x0055 );  /* 2nd cycle */
      FlashWrite( 0x5555L, 0x00A0 );  /* Program command */
      FlashWrite( ulOff, *uArrayPointer );   /* Program value */

      /* Step 5: Perform data polling until P/E.C. has completed. */
      /* See Data Polling Flowchart of the Data Sheet */
      if( FlashDataPoll( ulOff, *uArrayPointer ) == FLASH_POLL_FAIL )
      {
         FlashReadReset();
         return FLASH_PROGRAM_FAIL;
      }
```

(Cont'd in the next page)

```
      /* Step 6: Update pointers */
      ulOff++;          /* next word offset */
      uArrayPointer++;  /* next word in array */

   /* Step 7: End while loop */
   }

   /* Step 8: Return to Read Array mode */
   FlashWrite( 0x0000L, 0x00F0 ); /* Use single instruction cycle method */

   return FLASH_SUCCESS;
}



/*****************************************************************************
Function:     static int FlashDataPoll( unsigned long ulOff, unsigned int uVal )
Arguments:    ulOff should hold a valid offset to be polled. For programming
   this will be the offset of the word being programmed. For erasing this can
   be any address in the block(s) being erased.
   uVal should hold the value being programmed. A value of FFFFh should be used
   when erasing.
Return Value: The function returns FLASH_SUCCESS if the P/E.C. is successful
   or FLASH_POLL_FAIL if there is a problem.
Description: The function is used to monitor the P/E.C. during erase or
   program operations. It returns when the P/E.C. has completed. The Data Sheet
   gives a flow chart (Data Polling Flowchart) showing the operation of the
   function.

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

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

(Cont'd in the next page)

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

      /* Step 3: Else if DQ5 is zero then operation is not yet complete */
      if( (u&0x0020) == 0x0000 )              /* DQ5 == 0 (1 for Erase Error) */
         continue;

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

      /* Step 5: If DQ7 is now the same as Value(bit 7) then
         return FLASH_SUCCESS */
      if( (u&0x0080) == (uVal&0x0080) )       /* DQ7 == DATA  */
         return FLASH_SUCCESS;

      /* Step 6: Else return FLASH_POLL_FAIL */
      else                                /* DQ7 here means fail */
         return FLASH_POLL_FAIL;
   }
}


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

Description:  This function can only be called after an erase operation which
   has failed the FlashDataPoll() function. It must be called before the reset
   is made.
      The function reads bit 2 of the Status Register to determine is the block
   has erased successfully or not. Successfully erased blocks should have DQ2
   set to 1 following the erase. Failed blocks will have DQ2 toggle.
Pseudo Code:
   Step 1: Read DQ2 in the block twice
   Step 2: If they are both the same then return FLASH_SUCCESS
   Step 3: Else return FLASH_BLOCK_FAILED_ERASE
*****************************************************************************/
static int FlashBlockFailedErase( unsigned char ucBlock )
{
   int FirstRead, SecondRead; /* Two variables used for clarity, Optimiser will
                                 probably not use any */

   /* Step 1: Read block twice */
   FirstRead  = FlashRead( BlockOffset[ucBlock] ) & 0x0004;
   SecondRead = FlashRead( BlockOffset[ucBlock] ) & 0x0004;
```

(Cont'd in the next page)

```
   /* Step 2: If they are the same return FLASH_SUCCESS */
   if( FirstRead == SecondRead )
      return FLASH_SUCCESS;

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



/*****************************************************************************
Function:     char *FlashErrorStr( int ErrNum );
Arguments:    ErrNum 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 ErrNum )
{
   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 Out Of Range",
                          "Flash is Wrong Type",
                          "Flash Block Failed Erase" };

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

   /* Step 1,2 Return the correct string */
   if( ErrNum < 1 || ErrNum > 9 ) /* Check the range */
      return "Unknown Error\n";

   else
      return str[ErrNum-1];
}
```

(Cont'd in the next page)

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

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
Return Value: -2
Description:  The P/E.C. 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 fail a second time then it is
   likely that the device will need to be replaced.

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

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

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

Error Name:   FLASH_MPU_TOO_SLOW
Return Value: -4
Description:  The MPU has not managed to write all of the selected blocks to the
   device before the timeout period expired. See BLOCK ERASE (BE) INSTRUCTION
   section of the Data Sheet for details.
Solutions:    If this occurs occasionally then it may be because an interrupt is
   occuring during between writing the blocks to be erased. Seach for "DSI!" in
   the code and disable interrupts during the time critical sections.
   If this command 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.
```

(Cont'd in the next page)

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

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 sector and re-programming the value. If it fails
   again then the device may need to be changed.

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

Error Name:   FLASH_OFFSET_OUT_OF_RANGE
Return Value: -7
Description:  The offset given is out of the range of the device.
Solution:     Check the offset range 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
   SGS-THOMSON for a different source code library.

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

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

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

```
****M29F200B.H Header File for M29F200B.C***********************************

   Filename:    m29f200b.h
   Description: Header file for m29f200b.c. Consult the C file for details

   Copyright (c) 1997 SGS-THOMSON Microelectronics.

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

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

/*****************************************************************************
Error Conditions and return values.
See end of C file for explanations and help
*****************************************************************************/
#define FLASH_BLOCK_PROTECTED      (0x01)
#define FLASH_BLOCK_UNPROTECTED    (0x00)
#define FLASH_BLOCK_NOT_ERASED     (0xFF)
#define FLASH_BLOCK_ERASE_FAILURE  (0xFE)
#define FLASH_BLOCK_ERASED         (0xFD)
#define FLASH_SUCCESS              (-1)
#define FLASH_POLL_FAIL            (-2)
#define FLASH_TOO_MANY_BLOCKS      (-3)
#define FLASH_MPU_TOO_SLOW         (-4)
#define FLASH_BLOCK_INVALID        (-5)
#define FLASH_PROGRAM_FAIL         (-6)
#define FLASH_OFFSET_OUT_OF_RANGE  (-7)
#define FLASH_WRONG_TYPE           (-8)
#define FLASH_BLOCK_FAILED_ERASE   (-9)


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

(Cont'd in the next page)

```
extern char *FlashErrorStr( int ErrNum );

-----------------end of M29F200B.H -------------------------
```

```
/****M29F200B.C*2Mb Flash Memory*8bit mode************************************

   Filename:    m29f200b.c
   Description: Library routines for the M29F200 2Mbit (256kx8) Flash Memory.
                8 bit drivers.

   Version:     1.00
   Date:        27/8/97
   Author:      Brendan Watts, OTS & Alexandre Nairac
   Copyright (c) 1997 SGS-THOMSON Microelectronics.

   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   22/07/97  Initial Release of the Software.
   1.01   27/08/97  Minor revisions to the Software.

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

   This source file provides library C code for using the M29F200 devices (both
   M29F200T and M29F200B) in 8 bit mode. A separate file is available for users
   who wish to access the device in 16 bit mode.

   The following functions are available in this library:

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

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

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

      FlashWrite() for writing a word to the flash
```

(Cont'd in the next page)

```
      FlashRead()  for reading a word from the flash
      FlashPause() for timing short pauses (in micro seconds)

   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.

   C does not include a method for disabling interrupts to keep time-critical
   sections of code from being interrupted. The user may wish to disable
   interrupts during parts of the code to avoid the FLASH_MPU_TO_SLOW error
   from occuring if an interrupt occurs at the wrong time. Where interrupts
   should be disabled and re-enabled there is a /# DSI! #/ or /# ENI! #/
   comment.

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

#include <stdlib.h>

#include "m29f200b.h"        /* Header file with global prototypes */

#define USE_M29F200T


/*****************************************************************************
Constants
*****************************************************************************/
#define COUNTS_PER_MICROSECOND (200)
#define MANUFACTURER_ST (0x20)
#define BASE_ADDR ((volatile unsigned char*)0x0000)
   /* BASE_ADDR is the base address of the flash, see the functions FlashRead
      and FlashWrite(). Some applications which require a more complicated
      FlashRead() or FlashWrite() may not use BASE_ADDR */


#ifdef USE_M29F200T
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0 */
   0x10000L,  /* Start offset of block 1 */
   0x20000L,  /* Start offset of block 2 */
   0x30000L,  /* Start offset of block 3 */
   0x38000L,  /* Start offset of block 4 */
   0x3A000L,  /* Start offset of block 5 */
   0x3C000L   /* Start offset of block 6 */
};
```

(Cont'd in the next page)

```
#define EXPECTED_DEVICE (0xD3) /* M29F200T */
#endif

#ifdef USE_M29F200B
static const unsigned long BlockOffset[] =
{
   0x00000L,  /* Start offset of block 0 */
   0x04000L,  /* Start offset of block 1 */
   0x06000L,  /* Start offset of block 2 */
   0x08000L,  /* Start offset of block 3 */
   0x10000L,  /* Start offset of block 4 */
   0x20000L,  /* Start offset of block 5 */
   0x30000L   /* Start offset of block 6 */
};

#define EXPECTED_DEVICE (0xD4) /* M29F200B */

#endif

#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))
#define FLASH_SIZE (0x40000L) /* 256k x8 */




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

The following functions are only needed in this module.
*****************************************************************************/

static unsigned char FlashWrite( unsigned long ulOff, unsigned char ucVal );
static void FlashPause( unsigned int uMicroSeconds );
static int FlashDataPoll( unsigned long ulOff, unsigned char ucVal );
static int FlashBlockFailedErase( unsigned char ucBlock );


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

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

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

(Cont'd in the next page)

```
   The function is made to return ucVal so that it is compatible with the macro.

Pseudo Code:
   Step 1: Write ucVal to the byte offset in the flash
   Step 2: return ucVal

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



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

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

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

Pseudo Code:
  Step 1: Return the value at byte offset ulOff

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



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

(Cont'd in the next page)

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

Pseudo Code:
   Step 1: Compute count size for pause.
   Step 2: Count to the required size.

*************************************************************/
static void FlashPause( unsigned int uMicroSeconds )
{
   volatile unsigned long ulCountSize;

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

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



/*****************************************************************************
Function:      void FlashReadReset( void )
Arguments:     none
Return Value:  none
Description:   This function places the flash in the Read 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.

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

Pseudo Code:
   Step 1: write command sequence (see Instructions Table of the Data Sheet)
   Step 2: wait 10us

*****************************************************************************/
void FlashReadReset( void )
{
   /* Step 1: write command sequence */
   FlashWrite( 0xAAAAL, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x5555L, 0x55 );  /* 2nd Cycle */
```

(Cont'd in the next page)

```
   FlashWrite( 0xAAAAL, 0xF0 );  /* 3rd Cycle */

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

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

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

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

   When iFunc is FLASH_READ_DEVICE_CODE (-1) the function returns the Device
   Code.  The device codes for the parts are:
      M29F200T   D3h
      M29F200B   D4h

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

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

Pseudo Code:
   Step 1:  Send the Auto Select Instruction to the device
   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( 0xAAAAL, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x5555L, 0x55 );  /* 2nd Cycle */
   FlashWrite( 0xAAAAL, 0x90 );  /* 3rd Cycle */

   /* Step 2: Read the required function */
   if( iFunc == FLASH_READ_MANUFACTURER )
      iRetVal = (int) FlashRead( 0x0000L ); /* A0 = A1 = A6 = 0 */

   else if( iFunc == FLASH_READ_DEVICE_CODE )
```

(Cont'd in the next page)

```
      iRetVal = (int) FlashRead( 0x0002L );
                                        /* A0 = 1, A1 = A6 = 0, remember A-1 */

   else if( (iFunc >= 0) && (iFunc < NUM_BLOCKS) )
      iRetVal = (int) FlashRead( BlockOffset[iFunc] + 0x0004L );
                                      /* A0 = A6 = 0, A1 = 1, remember A-1 */
   else
      iRetVal = FLASH_BLOCK_INVALID;

   /* Step 3: Return to Read Array mode */
   FlashWrite( 0x0000L, 0xF0 ); /* Use single instruction cycle method */

   return iRetVal;
}
```

```
/*****************************************************************************
Function:     int FlashBlockErase( unsigned char ucNumBlocks,
   unsigned char ucBlock[]  )
Arguments:    ucNumBlocks holds the number of blocks in the array ucBlock
   ucBlock is an array containing the blocks to be erased.
Return Value: The function returns the following conditions:
   FLASH_SUCCESS              (-1)
   FLASH_POLL_FAIL            (-2)
   FLASH_TOO_MANY_BLOCKS      (-3)
   FLASH_MPU_TOO_SLOW         (-4)
   FLASH_WRONG_TYPE           (-8)
   Number of the first protected or invalid block

      The user's array, ucBlock[] is used to report errors on the specified
   blocks. If a time-out occurs because the MPU is too slow then the blocks
   in ucBlocks which are not erased are overwritten with FLASH_BLOCK_NOT_ERASED
   (FFh) and the function returns FLASH_MPU_TOO_SLOW.
      If an error occurs during the erasing of the blocks the blocks in
   ucBlocks which have failed the erase are set to FLASH_BLOCK_ERASE_FAILURE
   (FEh) and the function returns FLASH_POLL_FAIL.
      If both errors occur then the function will set the ucBlock array for
   each type of error (i.e. either to FLASH_BLOCK_NOT_ERASED or to
   FLASH_BLOCK_ERASE_FAILURE). It will return FLASH_POLL_FAIL even though
   the FLASH_MPU_TOO_SLOW has also occured.

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

(Cont'd in the next page)

```
  During the Erase Cycle the Data Polling Flowchart of the Data Sheet is
  followed. The toggle bit, DQ6, is not used. For an erase cycle the data on
DQ7
  will be '0' during the erase and '1' on completion.

Pseudo Code:
   Step 1:  Check for correct flash type
   Step 2:  Check for protected or invalid blocks
   Step 3:  Write Block Erase command
   Step 4:  Check for time-out blocks
   Step 5:  Wait for the timer bit to be set
   Step 6:  Perform Data Polling until P/E.C. has completed
   Step 7:  Return to Read Array mode
*****************************************************************************/
int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[] )
{
   unsigned char ucCurBlock;    /* Range Variable to track current block */
   int iRetVal = FLASH_SUCCESS; /* Holds return value: optimistic initially! */
   unsigned char FirstRead, SecondRead; /* used to check toggle bit DQ2 */

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

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

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

   /* Step 3: Write Block Erase command */
   FlashWrite( 0xAAAAL, 0xAA );
   FlashWrite( 0x5555L, 0x55 );
   FlashWrite( 0xAAAAL, 0x80 );
   FlashWrite( 0xAAAAL, 0xAA );
   FlashWrite( 0x5555L, 0x55 );
   /* DSI!: Time critical section. Additional blocks must be added every 80us */
   for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
   {
      FlashWrite( BlockOffset[ucBlock[ucCurBlock]], 0x30 );

      /* Check for Erase Timeout Period */
```

(Cont'd in the next page)

```
      if( (FlashRead( BlockOffset[ucBlock[0]] ) & 0x08) == 0x08 )
         break; /* Cannot set any more sectors due to timeout */
   }
   /* ENI! */

   /* Step 4: Check for time-out blocks */
   /* if timeout occured then check if last block is erasing or not */
   /* Use DQ2 of status register, toggle implies block is erasing */
   if( ucCurBlock < ucNumBlocks )
   {
      iRetVal = FLASH_MPU_TOO_SLOW;

      FirstRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x04;
      SecondRead = FlashRead( BlockOffset[ucBlock[ucCurBlock]] ) & 0x04;
      if( FirstRead != SecondRead )
      {
         ucCurBlock++; /* Point to the next block */
      }

      /* Now specify all other blocks as not being erased */
      while( ucCurBlock < ucNumBlocks )
      {
         ucBlock[ucCurBlock++] = FLASH_BLOCK_NOT_ERASED;
      }
   }

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

   /* Step 6: Perform data polling until P/E.C. completes, check for errors */
   if( FlashDataPoll( BlockOffset[ucBlock[0]], 0xFF ) != FLASH_SUCCESS )
   {
      for( ucCurBlock = 0; ucCurBlock < ucNumBlocks; ucCurBlock++ )
      {
         if( ucBlock[ucCurBlock] == FLASH_BLOCK_NOT_ERASED )
           break; /* The rest of the blocks have not been erased anyway */

         if( FlashBlockFailedErase( ucBlock[ucCurBlock] )
         == FLASH_BLOCK_FAILED_ERASE )
         {
            ucBlock[ucCurBlock] = FLASH_BLOCK_ERASE_FAILURE;
         }
      }
```

(Cont'd in the next page)

```
      iRetVal = FLASH_POLL_FAIL;
   }

   /* Step 7: Return to Read Array mode */
   FlashWrite( 0x0000L, 0xF0 ); /* Use single instruction cycle method */

   return iRetVal;
}
```

```
/*****************************************************************************
Function:     int FlashChipErase( unsigned char *Results )
Arguments:    Results is a pointer to an array where the results will be stored.
   If Results == NULL then no results are stored.
   Otherwise the results are written to the array if an error occurs. The
   array is left unchanged if the function returns FLASH_SUCCESS.
   The errors written to the array are:
      FLASH_BLOCK_ERASED (FDh)        if the block erased correctly
      FLASH_BLOCK_ERASE_FAILURE (FEh) if the block failed to erased
Return Value: On success the function returns FLASH_SUCCESS (-1)
   If a block is protected then the function returns the number of the block.
   If the erase algorithms fails then the function returns FLASH_POLL_FAIL (-2)
   If the wrong type of flash is accessed then the function returns
   FLASH_WRONG_TYPE (-8)
Description: The function can be used to erase the whole flash chip so long as
   no sectors are protected. If any sectors are protected then nothing is
   erased.
Pseudo Code:
   Step 1: Check for correct flash type
   Step 2: Check that all sectors are unprotected
   Step 3: Send Chip Erase Command
   Step 4: Perform data polling until P/E.C. has completed.
   Step 5: Check for blocks erased correctly
   Step 6: Return to Read Array mode
*****************************************************************************/
int FlashChipErase( unsigned char *Results )
{
   unsigned char ucCurBlock; /* Used to track the current block in a range */
   int iRetVal;      /* Holds the return value */

   /* 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;
```

(Cont'd in the next page)

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

   /* Step 3: Send Chip Erase Command */
   FlashWrite( 0xAAAA, 0xAA );
   FlashWrite( 0x5555, 0x55 );
   FlashWrite( 0xAAAA, 0x80 );
   FlashWrite( 0xAAAA, 0xAA );
   FlashWrite( 0x5555, 0x55 );
   FlashWrite( 0xAAAA, 0x10 );

   /* Step 4: Perform data polling until P/E.C. completed */
   iRetVal = FlashDataPoll( 0x0000, 0xFF );/* Erasing writes 0xFF to flash */

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

   /* Step 6: Return to Read Array mode */
   FlashWrite( 0x0000, 0xF0 ); /* Use single instruction cycle method */

   return iRetVal;
}


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

(Cont'd in the next page)

```
   If the address 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 accessed then the function returns
   FLASH_WRONG_TYPE (-8).
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 is not erased first.

Pseudo Code:
   Step 1: Check that the flash is of the correct type
   Step 2: Check the offset range is valid.
   Step 3: While there is more to be programmed
   Step 4:  Program the next byte
   Step 5:  Perform data polling until P/E.C. has completed.
   Step 6:  Update pointers
   Step 7: End of While Loop
   Step 8: Return to Read Array mode
*****************************************************************************/
int FlashProgram( unsigned long ulOff, size_t NumBytes, void *Array )
{
   unsigned char *ucArrayPointer;/* Use an unsigned char to access the array */
   unsigned long ulLastOff;      /* Holds the last offset to be programmed */

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

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

   /* Step 3: While there is more to be programmed */
   ucArrayPointer = (unsigned char *)Array;
   while( ulOff <= ulLastOff )
   {
      /* Step 4: Program the next byte */
      FlashWrite( 0xAAAAL, 0xAA );  /* 1st cycle */
      FlashWrite( 0x5555L, 0x55 );  /* 2nd cycle */
      FlashWrite( 0xAAAAL, 0xA0 );  /* Program command */
      FlashWrite( ulOff, *ucArrayPointer );   /* Program value */

      /* Step 5: Perform data polling until P/E.C. has completed. */
      /* See Data Polling Flowchart of the Data Sheet */
      if( FlashDataPoll( ulOff, *ucArrayPointer ) == FLASH_POLL_FAIL )
      {
         FlashReadReset();
         return FLASH_PROGRAM_FAIL;
      }
```

(Cont'd in the next page)

```
      /* Step 6: Update pointers */
      ulOff++;          /* next byte offset */
      ucArrayPointer++;  /* next byte in array */

   /* Step 7: End while loop */
   }

   /* Step 8: Return to Read Array mode */
   FlashWrite( 0x0000L, 0xF0 ); /* Use single instruction cycle method */

   return FLASH_SUCCESS;
}



/*****************************************************************************
Function:     static int FlashDataPoll( unsigned long ulOff,
                                        unsigned char ucVal )
Arguments:    ulOff should hold a valid offset to be polled. For programming
   this will be the offset of the byte being programmed. For erasing this can
   be any address in the block(s) being erased.
   ucVal should hold the value being programmed. A value of FFh should be used
   when erasing.
Return Value: The function returns FLASH_SUCCESS if the P/E.C. is successful
   or FLASH_POLL_FAIL if there is a problem.
Description: The function is used to monitor the P/E.C. during erase or
   program operations. It returns when the P/E.C. has completed. The Data Sheet
   gives a flow chart (Data Polling Flowchart) showing the operation of the
   function.

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

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

(Cont'd in the next page)

```
      uc = FlashRead( ulOff );              /* Read DQ5, DQ7 at valid addr */

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

      /* Step 3: Else if DQ5 is zero then operation is not yet complete */
      if( (uc&0x20) == 0x00 )             /* DQ5 == 0 (1 for Erase Error) */
         continue;

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

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

      /* Step 6: Else return FLASH_POLL_FAIL */
      else                                /* DQ7 here means fail */
         return FLASH_POLL_FAIL;
   }
}


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

Description:  This function can only be called after an erase operation which
   has failed the FlashDataPoll() function. It must be called before the reset
   is made.
      The function reads bit 2 of the Status Register to determine is the block
   has erased successfully or not. Successfully erased blocks should have DQ2
   set to 1 following the erase. Failed blocks will have DQ2 toggle.
Pseudo Code:
   Step 1: Read DQ2 in the block twice
   Step 2: If they are both the same then return FLASH_SUCCESS
   Step 3: Else return FLASH_BLOCK_FAILED_ERASE
*****************************************************************************/
static int FlashBlockFailedErase( unsigned char ucBlock )
{
   unsigned char FirstRead, SecondRead; /* Two variables used for clarity,
                                    Optimiser will probably not use any */

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

(Cont'd in the next page)

```
   SecondRead = FlashRead( BlockOffset[ucBlock] ) & 0x04;

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

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




/*****************************************************************************
Function:     char *FlashErrorStr( int ErrNum );
Arguments:    ErrNum 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 ErrNum )
{
   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 Out Of Range",
                          "Flash is Wrong Type",
                          "Flash Block Failed Erase" };

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

   /* Step 1,2 Return the correct string */
   if( ErrNum < 1 || ErrNum > 9 ) /* Check the range */
      return "Unknown Error\n";

   else
      return str[ErrNum-1];
}
```

(Cont'd in the next page)

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

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
Return Value: -2
Description:  The P/E.C. 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 fail a second time then it is
   likely that the device will need to be replaced.

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

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

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

Error Name:   FLASH_MPU_TOO_SLOW
Return Value: -4
Description:  The MPU has not managed to write all of the selected blocks to the
   device before the timeout period expired. See BLOCK ERASE (BE) INSTRUCTION
   section of the Data Sheet for details.
Solutions:    If this occurs occasionally then it may be because an interrupt is
   occuring during between writing the blocks to be erased. Seach for "DSI!" in
   the code and disable interrupts during the time critical sections.
   If this command 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.
```

(Cont'd in the next page)

```
   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 sector and re-programming the value. If it fails
   again then the device may need to be changed.

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

Error Name:   FLASH_OFFSET_OUT_OF_RANGE
Return Value: -7
Description:  The offset given is out of the range of the device.
Solution:     Check the offset range 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
   SGS-THOMSON for a different source code library.

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

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

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