AN945
APPLICATION NOTE

Software Drivers for M29F040 and M29W040 Flash Memories

CONTENTS

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 M29F040 and the M29W040 Flash memories. The M29W040, with a rated power supply of 3V, is a low voltage version of the M29F040 whose power supply is rated at 5V. The M29W040 has slightly slower access times than the M29F040, but the two devices are otherwise very similar. This application note supports both devices and all technical information about the M29F040 also applies to the M29W040, except where otherwise specified.

The application note includes listings of the source code which is also available in file form. The m29f040.c and m29f040.h files contain libraries for accessing both the M29F040 and the M29W040 Flash memories.

An overview of the programming model for the M29F040 and M29W040 is given. The programming differences between the M29F040 and AMD's Am29F040 are described. Advice on how to modify programs already written for AMD's Am29F040 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 M29F040 and M29W040.

This application note does not replace the M29F040 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 M29F040 PROGRAMMING MODEL

The M29F040 is a 512 Kbit x8 Flash memory which can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. The device is broken down into 8 blocks, each 64 Kbytes in size. Each block can be erased individually, or the whole chip can be erased at once, erasing all 4 Mbit.

The M29F040 is a single voltage device. It differs from first generation devices which require a 12V supply to program or erase. The M29F040 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 M29F040 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 M29F040 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 pre- programmed, protected and unprotected by an EPROM programmer prior to fitting into the hardware. Unprotected blocks may still be used to store data and parameters. By protecting a block, accidental data loss through software failure cannot occur.

MODES

All write accesses to the M29F040 go to the P/E.C. which decodes them as commands. These commands are used to put the M29F040 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 M29F040. 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 M29F040 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 M29F040 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 on the M29F040 or 1.5s on the M29W040.

The Instructions Table of the M29F040 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.

To change between modes write accesses to the specified addresses with the correct data are required. For example entering the program/erase mode and programming 65h to the address 03E2h requires the user to write the following sequence (in C):

```
*(unsigned char*)(0x5555) = 0xAA;
*(unsigned char*)(0x2AAA) = 0x55;
*(unsigned char*)(0x5555) = 0xA0;
*(unsigned char*)(0x03E2) = 0x65;
```

This example assumes that the M29F040 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 M29F040 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 M29F040 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 M29F040 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 Am29F040

If the user has already developed code which supports the Am29F040 devices then modifying the drivers to include the M29F040 should be straight forward.

In many cases there will be no changes required to the system software. The M29F040 and the Am29F040 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.

Table 1. Manufacturer and Device Codes Table

Device	Manufacturer Code	Device Code
M29F040	20h	E2h
Am29F040	01h	A4h

Consider the following example of code:

```
unsigned char device_code( void )
{
   /* Write auto select sequence */
   FlashWrite( 0x5555L, 0xAA );
   FlashWrite( 0x2AAAL, 0x55 );
   FlashWrite( 0x5555L, 0x90 );

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

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

In order to hide the difference between the M29F040 and the Am29F040 from the user s source code the function could be changed to the following:

```
unsigned char device_code( void )
{
  /* Write auto select sequence */
  FlashWrite( 0x5555L, 0xAA );
  FlashWrite( 0x2AAAL, 0x55 );
  FlashWrite( 0x5555L, 0x90 );

  /* Check for ST */
  if( FlashRead(0x0000L) == 0x20 )
  {

   if( FlashRead(0x0001L) == 0xA4)
      return 0xE2;
  }

  /* 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 Am29F040 and the M29F040 from the return value of `'device_code'`. For applications, such as flash programmers, which need to distinguish between STMicroelectronics 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 STMicroelectronics parts.

GENERATING STMicroelectronics CODE

Low-level functions (drivers) have been provided to simplify the process of developing application code in C for the STMicroelectronics Flash memories (M29F040 and M29W040). 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 flash can be specified by a 32 bit linear pointer or a 32 bit offset into the device could be provided by the user. The FlashRead() functions in each case would declared as:

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

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

The pointer option has the advantage that it runs faster. The 32 bit offset needs to be changed to an address for each access and this involves 32 bit arithmetic.

Using a 32 bit offset is, however, more portable since the resulting software can easily be changed to run on microprocessors with segmented memory spaces (such as the 8086).

For maximum portability all the functions in this application note use a 32 bit `unsigned long` offset, rather than a pointer.

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_M29F040 or USE_M29W040 depending on whether an M29F040 or an M29W040 is fitted. The top of the source file defines M29F040.
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 M29F040. 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 M29F040'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.

Users should be aware of the differences between the Am29F040B and the M29F040/Am29F040. Three additional features have been added to the AMD part, these are:

1. The DQ2 toggle bit can be used to tell which blocks are being erased.
2. The user can program blocks which are not being erased during the erase suspend mode.
3. When writing command sequences to the Am29F040B the address lines A11 to A18 can be set to any value, whereas for the M29F040/ Am29F040B A11 to A15 need to be set correctly, A16 to 18 can be any value.

The source code in this application note will also work with the new part. However, care should be taken when fitting an M29F040/Am29F040 in place of an Am29F040B in case the new features have been used in the software.

CONNECTION TO COMMON MICROPROCESSORS

The M29F040 and M29W040 can be connected easily to a variety of microprocessors. In the examples given here the connection of the M29F040 to the ST10R163 shows how it can be used with a non-multiplexed bus whereas the i80960SA shows how to connect it to a multiplexed bus. The M29W040 is shown connected to the MC68331 (non-multiplexed).

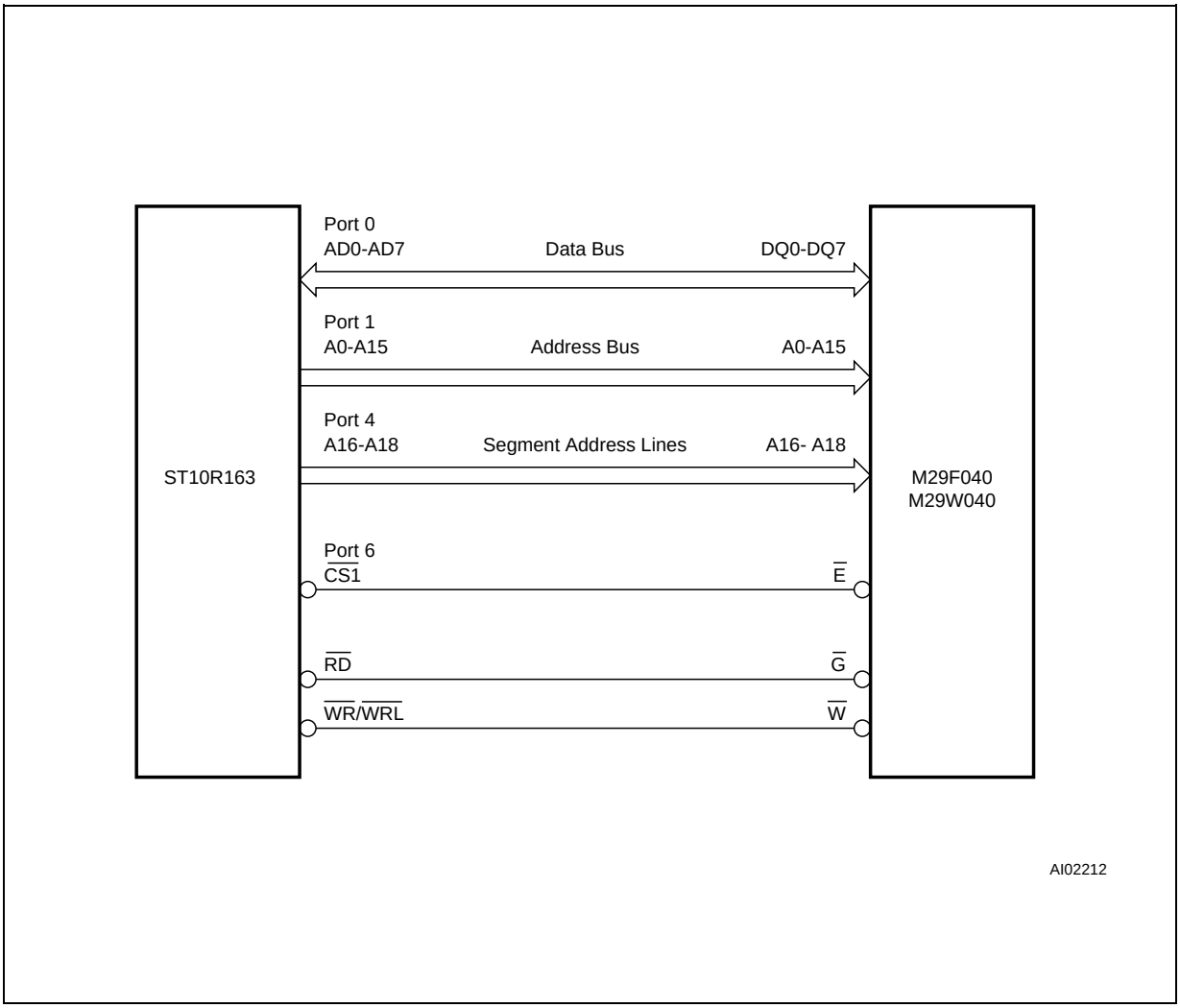
Connection to the ST10R163

Figure 1 shows an example of how to connect the ST10R163 to the M29F040. In this example the ST10R163 accesses the M29F040 in 8-bit non- multiplexed bus mode.

The Port6 $\overline{CS1}$ output of the ST10R163 is used to enable the M29F040. The address range over which $\overline{CS1}$ enables the M29F040 is selected by configuring BUSCON1 of the ST10R163. Note that if other parts of the system run in 16-bit mode then BUSCON1 will need to be configured for 8-bit operation.

The data bus of the M29F040 is connected to the Least Significant Byte of the ST10R163. The address bus is connected to the address bus of the ST10R163 with the upper bits of the ST10R163 remaining unconnected.

Figure 1. Connection of the M29F040 and M29W040 to the ST10R163



Connection to the Intel 80960SA

Figure 2 shows an example of how to connect an i80960SA to the M29F040. 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 M29F040 ($\overline{G}$, $\overline{E}$ and $\overline{W}$); these must be generated by external logic.

To make matters more confusing the i80960SA only uses 16-bit accesses. To solve this problem the M29F040 needs to occupy 1M Byte of address space. Only even addresses will access the flash. Odd addresses will not access anything.

The Least Significant Byte of the i80960SA s data bus should be connected to the data bus of the M29F040, the Most Significant Byte should remain unconnected.

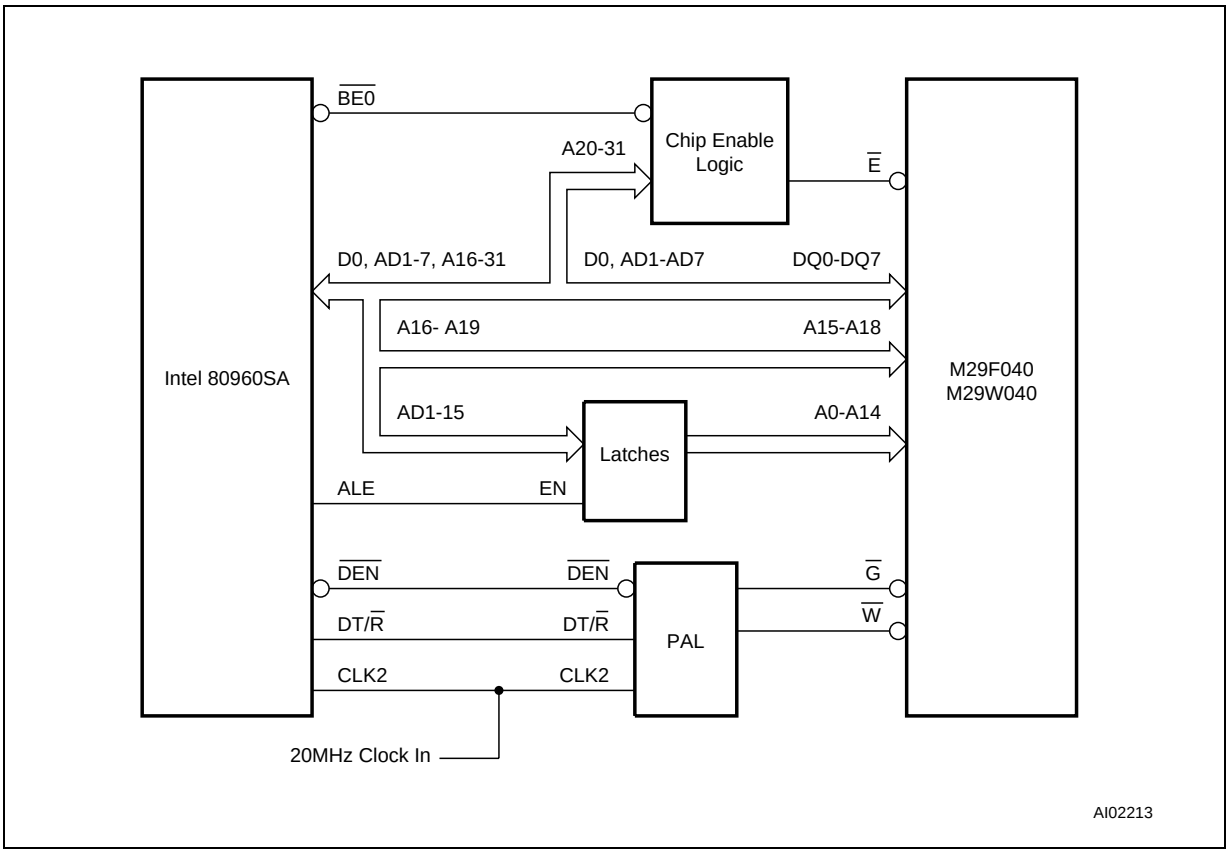
Systems which use multiple flash devices can take advantage of the odd addresses by connecting a second M29F040 to the Most Significant Data lines of the i80960SA. Care should be taken using this type of system since a write will always access both flash devices unless the $\overline{BE0:1}$ signals are decoded.

The chip enable logic in the example can be made to map the 512K Bytes memory space of the M29F040 to any 1M byte boundary of the 4G Byte address space of the i80960SA.

The least significant address bit of the i80960SA is A1 and therefore the AD1 signal of the i80960SA corresponds to A0 of the M29F040. 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 M29F040.

Figure 2. Connection of the M29F040 and M29W040 to the 80960SA



The PAL provides the control signals which drive the M29F040. This example provides zero wait state access to an M29F040-70 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 M29F040 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 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 M29F040 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 M29F040 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.

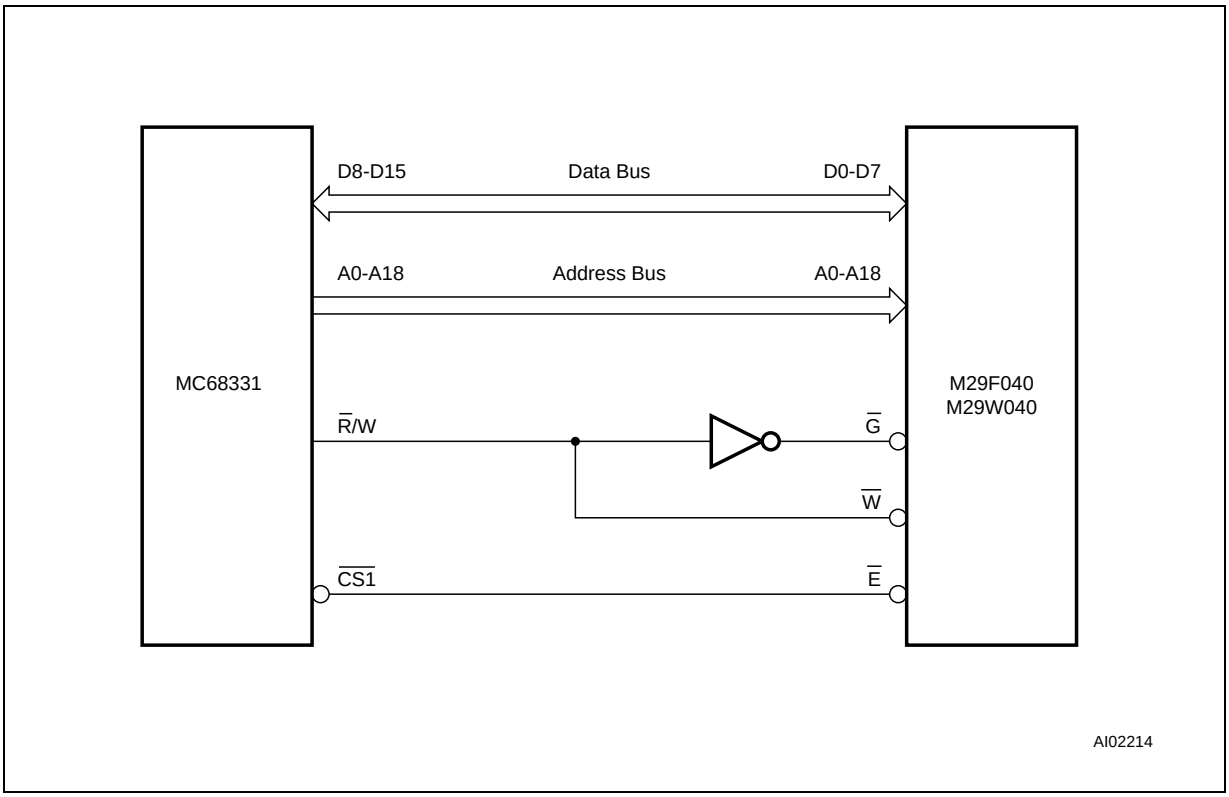
Connection to the MC68331

The MC68331 has a non-multiplexed 16-bit data bus which can be configured for 8-bit access. 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 M29W040 is connected to D8 of the MC68331, with the other data lines following.

The MC68331 has eleven chip selects, one of which can readily be used to select the M29W040 device. Furthermore 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 M29W040.

The MC68331 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 M29W040. During the read cycle the MC68331 does not require the data to be held after $\overline{CS1}$ goes high, allowing the M29W040 $\overline{E}$ input to be driven directly from $\overline{CS1}$.

Figure 3. Connection of the M29F040 and M29W040 to the MC68331



CONCLUSION

The M29F040 and M29W040 single voltage Flash memories are ideal products for embedded and other computer systems, able to be easily interfaced to microprocessors and driven with simple software drivers written in the C language.

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

```
/****m29f040.h Header File for m29f040.c*************************************

   Filename:    m29f040.h
   Description: Header file for m29f040.c. Consult the C file for details

   Copyright (c) 1997 STMicroelectronics.

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


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



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

/*****************************************************************************
Error Conditions and return values.
See end of C file for explanations and help
*****************************************************************************/
#define FLASH_BLOCK_PROTECTED      (0x01)
#define FLASH_BLOCK_UNPROTECTED    (0x00)
#define FLASH_BLOCK_NOT_ERASED     (0xFF)
#define FLASH_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_ADDRESS_OUT_OF_RANGE (-7)
#define FLASH_WRONG_TYPE           (-8)


/*****************************************************************************
Function Prototypes
*****************************************************************************/
extern void FlashReadReset( void );
extern unsigned char FlashRead( unsigned long Off );
extern int FlashAutoSelect( int iFunc );
```

(Cont'd in the next page)

```
extern int FlashBlockErase( unsigned char ucNumBlocks, unsigned char ucBlock[]);
extern int FlashChipErase( void );
extern int FlashProgram( unsigned long Off, size_t NumBytes, void *Array );
extern char *FlashErrorStr( int ErrNum );
```

```
/****M29F040.C*4Mb Flash Memory************************************************

   Filename:    m29f040.c
   Description: Library routines for the M29F040 4Mb (512k x8) Flash Memory.

   Revision:    1.00
   Date:        30/8/97
   Author:      Brendan Watts, OTS
   Copyright (c) 1997 STMicroelectronics.

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

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

   Version History.

   Ver.   Date      Comments

   1.00   30/08/97  Initial Release of the Software.

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

   The following functions are available in this library:

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

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

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

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


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

(Cont'd in the next page)

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

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

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

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

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

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

#include <stdlib.h>


#include "m29f040.h"        /* Header file with global prototypes */


#define USE_M29F040




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

#ifdef USE_M29F040
#define EXPECTED_DEVICE (0xE2) /* Device code for the M29F040 */
#endif
```

(Cont'd in the next page)

```
#ifdef USE_M29W040
#define EXPECTED_DEVICE (0xE3) /* Device code for the M29W040 */
#endif



static const unsigned long BlockOffset[] = /* Offset from BASE_ADDR of blocks */
{
   0x00000L,  /* Start offset of block 0 */
   0x10000L,  /* Start offset of block 1 */
   0x20000L,  /* Start offset of block 2 */
   0x30000L,  /* Start offset of block 3 */
   0x40000L,  /* Start offset of block 4 */
   0x50000L,  /* Start offset of block 5 */
   0x60000L,  /* Start offset of block 6 */
   0x70000L   /* Start offset of block 7 */
};

#define NUM_BLOCKS (sizeof(BlockOffset)/sizeof(BlockOffset[0]))
#define FLASH_SIZE (0x80000L) /* 512K */



/*****************************************************************************
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 );

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

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

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

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

(Cont'd in the next page)

```
   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 the byte offset into 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.

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

Pseudo Code:
```

(Cont'd in the next page)

```
   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 5us is required if the command is called during a program or
   erase instruction. This is included here to guarentee operation. The
   functions in the data sheet call this function if they suspect an error
   during programming or erasing so that the 5us 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 5us

*****************************************************************************/
void FlashReadReset( void )
{
   /* Step 1: write command sequence */
   FlashWrite( 0x5555L, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x2AAAL, 0x55 );  /* 2nd Cycle */
   FlashWrite( 0x5555L, 0xF0 );  /* 3rd Cycle */

   /* Step 2: wait 5us */
   FlashPause( 5 );
```

(Cont'd in the next page)

```
}

/*****************************************************************************
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 READ BLOCK PROTECTION (RBP) 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 code for the M29F040 is E2h.

   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 Read Electronic Signature instruction to the device
            (Note: this is the same as the Read Block Protection instruction)
   Step 2:  Read the required function from the device.
   Step 3:  Return the device to Read Array mode.
*****************************************************************************/
int FlashAutoSelect( int iFunc )
{
   int iRetVal; /* Holds the return value */

   /* Step 1: Send the Read Electronic Signature instruction */
   FlashWrite( 0x5555L, 0xAA );  /* 1st Cycle */
   FlashWrite( 0x2AAAL, 0x55 );  /* 2nd Cycle */
   FlashWrite( 0x5555L, 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 )
      iRetVal = (int) FlashRead( 0x0001L ); /* A0 = 1, A1 = A6 = 0 */

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

(Cont'd in the next page)

```
   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 function returns
   FLASH_POLL_FAIL.
      If both errors occur then the function will set the ucBlock array to
   FLASH_BLOCK_NOT_ERASED for the unerased blocks. It will return
   FLASH_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.

   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
```

(Cont'd in the next page)

```
   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! */

   /* 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, 0xAA );
   FlashWrite( 0x2AAAL, 0x55 );
   FlashWrite( 0x5555L, 0x80 );
   FlashWrite( 0x5555L, 0xAA );
   FlashWrite( 0x2AAAL, 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 */
      if( (FlashRead( BlockOffset[ucBlock[0]] ) & 0x08) == 0x08 )
         break; /* Cannot set any more sectors due to timeout */
   }
   /* ENI! */

   /* Step 4: Mark timed-out blocks */
   if( ucCurBlock < ucNumBlocks )
   {
      iRetVal = FLASH_MPU_TOO_SLOW;
```

(Cont'd in the next page)

```
      /* Specify all other blocks as not being erased */
      /* Note that we cannot tell if the first one is erasing or not */
      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 )
   {
      iRetVal = FLASH_POLL_FAIL;
   }

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

   return iRetVal;
}
```

```
/*****************************************************************************
Function:     int FlashChipErase( void )
Arguments:    none
Return Value: On success the function returns FLASH_SUCCESS (-1)
   If a block is protected then the function returns the number of the block and
   no blocks are erased.
   If the erase algorithms fails then the function returns FLASH_POLL_FAIL (-2)
   If the wrong type of flash is detected then FLASH_WRING_TYPE (-8) is
   returned.
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
```

(Cont'd in the next page)

```
   Step 3: Send Chip Erase Command
   Step 4: Perform data polling until P/E.C. has completed.
   Step 5: Return to Read Array mode
*****************************************************************************/
int FlashChipErase( void )
{
   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;

   /* 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( 0x5555L, 0xAA );
   FlashWrite( 0x2AAAL, 0x55 );
   FlashWrite( 0x5555L, 0x80 );
   FlashWrite( 0x5555L, 0xAA );
   FlashWrite( 0x2AAAL, 0x55 );
   FlashWrite( 0x5555L, 0x10 );

   /* Step 4: Perform data polling until P/E.C. completed */
   iRetVal = FlashDataPoll( 0x0000L, 0xFF );/* Erasing writes 0xFF to flash */

   /* Step 5: Return to Read Array mode */
   FlashWrite( 0x0000L, 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)
   If the address exceeds the address range of the Flash Device the function
   returns FLASH_ADDRESS_OUT_OF_RANGE (-7) and nothing is programmed.
```

(Cont'd in the next page)

```
   If the wrong type of flash is detected then FLASH_WRONG_TYPE (-8) is
   returned.
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 for correct flash type
   Step 2: Check the offset range is valid.
   Step 3: While there is more to be programmed
   Step 4:  Program the next 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 LastOff; /* 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 offset and range are valid */
   LastOff = ulOff+NumBytes-1;
   if( LastOff >= FLASH_SIZE )
      return FLASH_ADDRESS_OUT_OF_RANGE;

   /* Step 3: While there is more to be programmed */
   ucArrayPointer = (unsigned char *)Array;
   while( ulOff <= LastOff )
   {
      /* Step 4: Program the next byte */
      FlashWrite( 0x5555L, 0xAA );  /* 1st cycle */
      FlashWrite( 0x2AAAL, 0x55 );  /* 2nd cycle */
      FlashWrite( 0x5555L, 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;
      }

      /* Step 6: Update pointers */
```

(Cont'd in the next page)

```
      ulOff++;
      ucArrayPointer++;

   /* 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 Value )
Arguments:    ulOff should hold a valid offset to be polled. For programming
   this will be the offset of the byte being programmed. For erasing this can
   be any offset in the block(s) being erased.
   Value 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. in the M29F040
   Data Sheet th Data Polling Flow Chart shows 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 != 0), 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 Off, unsigned char Value )
{
   unsigned char uc;                /* holds value read from valid address */

   while( 1 )  /* TimeOut!: If, for some reason, the hardware fails then this
                  loop may not exit. Use a timer function to implement a timeout
                  from the loop. */
   {
      /* Step 1: Read DQ5 and DQ7 (into a byte) */
      uc = FlashRead( Off );              /* 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( (uc&0x80) == (Value&0x80) )       /* DQ7 == DATA  */
         return FLASH_SUCCESS;

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

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

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

      /* Step 6: Else return FLASH_POLL_FAIL */
      else                                /* DQ7 here means fail */
         return FLASH_POLL_FAIL;
   }  /* end of while loop */
}

/*****************************************************************************
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" };

   /* 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 > 8 ) /* Check the range */
```

(Cont'd in the next page)

```
      return "Unknown Error\n";

   else
      return str[ErrNum-1];
}



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

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.
Solution:     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 M29F040 Data Sheet for details.
Solution:     If this occurs occasionally then it may be because an interrupt is
   occuring during the writing the blocks to be erased. Seach for "DSI!" in
   the code and disable interrupts during the time critical sections.
```

(Cont'd in the next page)

```
            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.

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

Error Name:   FLASH_BLOCK_INVALID
Return Value: -5
Description:  A request for an invalid block has been made. Valid blocks range
   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.
Solution:     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_ADDRESS_OUT_OF_RANGE
Return Value: -7
Description:  The address given is out of the range of the device.
Solutions:    Check the address 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.
Solution:     Use a different flash chip with the target hardware or contact
   STMicroelectronics for a different source code library.

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

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

ask.memory@st.com

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