AN999
APPLICATION NOTE

Software Drivers for the M39432 FLASH+™ Multiple Memory

This document describes the primitive software drivers for use with the M39432. (The M39432 is a single-chip memory device containing a 4 Mb block of single-voltage Flash memory, and a 256 Kb block of Parallel EEPROM.) These routines have been developed on an evaluation board that was configured as follows:

- 1 ST10 microcontroller running at 40 MHz (compatible with SIEMENS 80C16X)
- 2 Mb SRAM (256 Kx8) occupying addresses 0H to 03FFFFH
- 1 M39432
  - EEPROM (32 Kx8) occupying addresses 40000H to 47FFFH
  - FLASH (512 Kx8) occupying addresses 80000H to FFFFFH
- 1 asynchronous serial interface for communication with a PC.

Once compiled and linked, the software routines were down-loaded to the static RAM (SRAM) of the evaluation board, via the asynchronous serial interface. A high level language debugger (HLLD) was then run on the processor of the evaluation board.

These routines were written in the C programming language, and can be easily adapted to many other different processor environments.

Most of the routines are extremely simple, but involve multiple operations on the device. For instance, to write a byte of data to the Flash memory area, the byte itself must be preceded by the “PROGRAM BYTE” command (hexadecimal AA,55,A0 written to three specific addresses). For the command to be recognised, the consecutive writes must be within a certain timing window of each other. The time between consecutive writes is called tWHWH, and must be no shorter than 0.2 µs (for physical reasons), and no longer than 80 µs (otherwise the command would be timed-out by an internal timer).

The sections that follow take each of the routines in turn, describing the constraints, and showing the C program solution.

Routine to Read a Byte from the M39432 Device

As indicated above, the EEPROM and Flash memory areas of the M39432 are mapped to distinct areas of the address space. For example:

- EEPROM addresses run from 40000H to 47FFFH
- FLASH addresses run from 80000H to FFFFFH

The Byte-Read operation is performed in the conventional manner, just as it would for a byte at any other address in main memory.

```
function READ(address);
  begin
    READ = *address
  end
```

Routine to Write a Byte to the M39432 Device

Similarly, there is a Write-Byte operation, for sending a byte of data or command to the device.

The write cycle time of the M39432 is guaranteed to be less than 10 ms. Even so, since the microcontroller is capable of executing a large number of instructions in this time, it might be desirable to call a CONTROL function to set internal control flags, thereby registering that a write operation is in progress, and allowing concurrent execution to be planned accordingly.

```
function WRITE(address,byte)
  begin
    *address = byte
    CONTROL ( if implemented )
  end
```

However, it must be emphasised that writing a byte to an address in the EEPROM or Flash memory area does not necessarily lead to the byte being stored at the given location. A central idea of non-volatile memory is that the device may be protected from casual write operations.

The above function is used for sending command bytes to the M39432 device, to put it in a specific mode of operation, as well as for sending data bytes to it, for storage. One of these commands is discussed next: the command for writing a byte in the Flash memory area.

Routine to Program a Single Byte in the Flash Memory Area

To write a single byte to the Flash memory, three command bytes (hexadecimal AA,55,A0) must first be written to three specific addresses. This is then followed by the byte that is to be written to the desired address. A function for doing this appears as follows:

```
function BYTE_WRITE_FLA (address,byte)
  begin
    WRITE (@05555h,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,A0H)
    WRITE (@address,byte)
    CONTROL ( if implemented )
  end
```

Once the final write operation has been initiated, the Flash memory takes over its own internal management, and will not accept any further operations until the previous one is completed.

Routine to Write a Byte in the EEPROM Area

The writing of a byte in the EEPROM area can be made to resemble the writing of a byte in RAM, or the writing of a byte in Flash memory, depending on the level of protection required. When the software data protection (SDP) mode is disabled, writing a byte into the EEPROM area is simply a matter of executing a single WRITE operation, giving the address of the desired cell within the EEPROM area. The function to do this is shown earlier, but is repeated here for completeness.

```
function WRITE(address,byte)
  begin
    *address = byte
    CONTROL ( if implemented )
  end
```

When the SDP mode is enabled, writing a byte into the EEPROM area resembles the process of programming a byte in the Flash memory area, as shown in the following function:

```
function SDP_BYTE(address,byte)
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,A0H)
    WRITE (address,byte)
    CONTROL ( if implemented )
  end
```

Routines to Enable and Disable Software Data Protection of the EEPROM Area

The purpose of the software data protection (SDP) mode is to offer some defence against inadvertent writes to the EEPROM area. When the mode is enabled, bytes in the EEPROM area can only be overwritten if the data write is preceded by the appropriate three byte command.

By default, the M39432 is delivered in the unprotected mode. The routine to enable SDP involves sending the (AA,55,A0) command on its own, as shown in the following function:

```
function SDP
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,A0H)
  end
```

The routine to disable SDP involves sending a six byte command (hexadecimal AA,55,80,AA,55,20) as shown in the following function:

```
function DIS_SDP
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,80H)
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,20H)
  end
```

Routine to Erase the Whole Flash Memory Area

The execution of the following six specific write operations (hexadecimal AA,55,80,AA,55,10) causes a bulk erase of the whole Flash memory.

```
function BULK
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,80H)
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,10H)
    CONTROL ( if implemented )
  end
```

Routine to Erase a Sector of the Flash Memory Area

The Flash memory is divided into eight sectors, each one 64 Kbytes in size. These can be erased together, as indicated in the previous section, or independently, as shown in this section. The operation involves the writing of five specific bytes (hexadecimal AA,55,80,AA,55) to put the device into the command mode, followed by a write of the value 30H to any address in each sector that is to be erased. An internal timer in the M39432 allows the device to recognise the end of the sector-erase operation: the device reverts back to normal mode when the last write operation is followed by a period of silence of 80 µs, or more. (The routine to handle the timer control is described in the next section.)

```
function SECTOR_ERASE (sector_address)
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@05555H,80H)
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (@sector_i_address,30H)
    WRITE (@sector_j_address,30H)
    /* and so on for each sector to be erased */
    CONTROL ( if implemented )
  end
```

Writing B0H at any address in the Flash memory area will suspend the erase operation.

Routing to Handle the Timer Control

The DQ3 status bit, known as the *erase time-out flag*, is reset when a sector-erase operation has been initiated on the device. This can be read, and used, by the CONTROL function:

a. checking that DQ3 is still reset, indicating that the device is still receptive of erase commands (the command 30H, sent to an address in the chosen sector, as described in the previous section)

b. checking that DQ3 has reverted to being set, indicating that the device has finished processing the last erase operation at the end of the SECTOR_ERASE function (as described in the previous section).

An example of the handling of this status bit is given in the following program fragment:

```
begin
  flag_timer= 1
  count= 0
  do
    begin
      read DQ3
      count= count+1
    end
  while (DQ3==0) and (count =< value)
  if count =< value then flag_timer= 0 (sector erase is processed, no more
    sector addresses can be sent)
  else flag_timer remains 1 (memory fail)
  return flag_timer
end
```

Routine to Handle the EEPROM Toggle-bit Control

When a new value has been written to the EEPROM area, there is a minimum time before the memory can accept the next operation. This delay can be hidden by allowing the processor to get on with other useful work, and later to poll the EEPROM to find out if it is still busy. The DQ6 status bit, known as the *toggle flag*, toggles between 0 and 1 on successive reads, until the erase/write cycle is complete. The state of completion, therefore, is indicated when two successive read operations of DQ6 return the same value.

The following program fragment implements the required algorithm:

```
begin
  n_bit= 6
  count6= 0
  flag_toggle= 1
  do
    begin
      first read of DQ6
      second read of DQ6
      count6= count6+1
    end
  while (first read of DQ6 != second read of DQ6) and (count6 =< max)
  if min=<count6=<max then flag_toggle= 0 (byte written OK)
  else flag_toggle= 1 (memory fail)
end
```

The completion of EEPROM erase/write cycles can alternatively be detected by controlling the DQ7 status bit, by a process known as *data polling*, as described next.

Routine to Handle the Data Polling Bit Control of the Flash Memory

The DQ7 status bit, known as the *data polling flag*, is an indicator for the end of byte-program and sector-erase operations in the Flash memory area. Data polling is effective after the fourth W pulse (for programming) or after the sixth W pulse (for erase). It must be performed at the address being programmed or at an address within the Flash sector being erased. When erasing or programming the Flash memory area, or when writing to the EEPROM area, bit DQ7 is held at the complement of the value that the processor wrote to it. Once the operation is completed, the bit is complemented again, back to its correct value.

In the following program fragment, the DQ5 status bit, known as the error flag, is also checked. A logic level '1' on this bit indicates a failure of the byte-programming, sector-erase or a bulk-erase operation. Any sector, in which an error of this type has occurred, must not be used any more. Other sectors may continue to be used.

```
begin
  flag_data_polling=1
  last_bit_dq7=last_byte&0x80 (bit 7 to be written in FLASH)
  do
    begin
      read DQ7
      read DQ5
    end
  while (DQ7!=last_bit_dq7) and (error_bit_dq5=0)
  if error_bit_DQ5=0 then flag_data_polling=0
  else
    begin
      read DQ7
      if data_polling_dq7=last_bit_dq7
      then flag_data_polling=0
      else
        begin
          flag_toggle=1
          RESET
        end
    end
  return flag_data_polling
end
```

When set, the error bit must be reset with the *reset* or *short reset* instruction (as described in the next section).

Routine to Reset the Flash Memory

This function must be run each time an error is detected on the error bit of the status register. It involves the writing of three specific bytes (hexadecimal AA,55,F0). A minimum delay of 5 µs is required before the next operation can be performed on the memory.

```
void RST
  begin
    WRITE (@05555H,AAH)
    WRITE (@02AAAH,55H)
    WRITE (address,F0)
    wait 5us
  end
```

Writing F0H at any address in the Flash memory area, known as a *short reset*, is equivalent to the above sequence.

```
void SHORT_RST
  begin
    WRITE (address,F0)
    wait 5us
  end
```

FURTHER READING

Table 1 lists the application notes that introduce ST's FLASH+ technology and the M39432 device.

Table 1. Bibliography of the Application Notes on FLASH+ Technology

Application Note	Content	Title
AN997	Introduction	M39432: a FLASH+™ Multiple Memory Device
AN931	Greater detail	On-Chip Hardware EEPROM Emulation versus Flash Memory Software Solutions
AN996	Glossary	Emulating the Unification of Flash and EEPROM: a Glossary
AN998	Technology	FLASH+™ Multiple Memory Technology
AN999	Drivers	Software Drivers for the M39432 FLASH+™ Multiple Memory

SOFTWARE EXAMPLES

The following three program fragments show examples of how the drivers can be used. The source files can be downloaded from the server. They are respectively: SCEE9701.C, SCEE9702.C and SCEE9703.C.

Example of a Program Using the Bulk Erase Function

This program fragment erases all the Flash memory sectors by using the *bulk erase function*. It also demonstrates the use of the RESET and CONTROL_DATA_POLLING_FLA functions.

```
/* FLASH+ ; M39432 DRIVER SOURCE CODE : FLASH BULK ERASE                    */
/* Version: 1.01                                                            */

/**************************************************************************/
/*         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.                                                   */
/**************************************************************************/

/*************************************************************/
/*                                                           */
/*      This program executes the bulk erase function.       */
/*      All the sectors in the flash will be erased.         */
/*                                                           */
/*      input variable :        none                         */
/*      internal variable :     bulk_ok,=0 when the bulk     */
/*                              erase is ok                  */
/*      output variable :       none                         */
/*                                                           */
/*                                                           */
/*************************************************************/
#define byte1 0xaa
#define byte2 0x55
#define byte_erase 0x80
#define byte_confirm 0x10
#define byte_rst 0xf0

#define prog_code1 0x85555
#define prog_code2 0x82aaa
```

```
                        /* DECLARATION OF FUNCTIONS */

 /* write and read functions*/
 void WRITE(char xhuge *,char );
 char READ_BIT(char, char xhuge * );
 char READ(char xhuge *addr);

 /* erase function*/
 char BULK_ERASE(void);

 /* control function*/
char CONTROL_DATA_POLLING_FLA(char,char xhuge *);

 /* reset function*/
 void RST(char xhuge *);

                        /* MAIN STARTS HERE */
 void main (void)   {
 char bulk_ok=1;

        bulk_ok = BULK_ERASE();
}
                        /* END OF MAIN */

/**FUNCTION BULK_ERASE: FLASH***********************************/
/*                                                             */
/*      input variable :        none                           */
/*                                                             */
/*      output variable :       flag_polling_dq7,receives the  */
/*                              result of the data polling, if */
/*                              this flag=0 the erase is       */
/*                              complete.                      */
/*  this function issues the bulk erase command                */
/*  all the sectors in the flash will be erased                */
/*************************************************************/

char BULK_ERASE(void)  {

 char xhuge *addr;
 char byte, flag_polling_dq7=1,last_byte=0xff;

/* coded cycle */
 addr= prog_code1;/* write 0xaa at @5555H IN FLASH (base=80000) */
 byte=byte1;
```

```
 WRITE(addr,byte);

 addr= prog_code2;/* write 0x55 at @2AAAH IN FLASH (base=80000) */
 byte=byte2;
 WRITE(addr,byte);

/*  erase command */
 addr=prog_code1;/* write 0x80 at @5555H IN FLASH (base=80000)  */
 byte=byte_erase;
 WRITE(addr,byte);

/* coded cycle */
 addr= prog_code1;/* write 0xaa at @5555H IN FLASH (base=80000) */
 byte=byte1;
 WRITE(addr,byte);

 addr= prog_code2;/* write 0x55 at @2AAAH IN FLASH (base=80000) */
 byte=byte2;
 WRITE(addr,byte);

/*bulk erase confirm */
 addr=prog_code1;/* write 0x10 at @5555H IN FLASH (base=80000)  */
 byte=byte_confirm;
 WRITE(addr,byte);
 flag_polling_dq7= CONTROL_DATA_POLLING_FLA(last_byte,addr);
 return(flag_polling_dq7);
}

/**FUNCTION WRITE: TO WRITE A BYTE IN FLASH MEMORY***************/
/*                                                               */
/*      input variable :        addr, write address              */
/*                              byte                             */
/*      output variable :       none                             */
/*                                                               */
/*  this function writes a byte at the given address             */
/*****************************************************************/

void WRITE(char xhuge *addr,char byte)  {

        *addr=byte;
}

/**FUNCTION READ: TO READ A BYTE FROM FLASH MEMORY***************/
/*                                                               */
```

```
/*      input variable :        addr, read address              */
/*                                                              */
/*      output variable :       read, read byte                 */
/*                                                              */
/*  this function reads a byte at the given address             */
/**************************************************************/

char READ(char xhuge *addr) {
char read;

      read = *addr;
      return(read);
}

/**FUNCTION CONTROL_DATA_POLLING_FLA (FLASH)*******************/
/*                                                              */
/*      input variable :        addr, read address              */
/*                              last_byte, last byte written    */
/*                                                              */
/*  internal variables :        read bits from the status       */
/*                              register:data_polling_bit_dq7   */
/*                                                              */
/*  When a programming operation is in progress, this bit       */
/*  outputs the complement of the bit being programmed on dq7.  */
/*  During an Erase operation, it outputs '0' then '1' after    */
/*  Erase completion.                                           */
/*                                                              */
/*  error_bit_dq5 outputs 1 when there is an error              */
/*  during programming or erasing.                              */
/*                                                              */
/*   last_bit_dq7 variable which stores the value of the last   */
/*   bit dq7 being programmed is set to 0FFH before the erasing.*/
/*   n_bit, bit number to be read.                              */
/*                                                              */
/*   output variable :  flag_data_polling, returns 0 when       */
/*                      the read bit = last_bit written         */
/*                      or when dq7=1 for erasure.              */
/*                      It returns 1 if :                       */
/*                      - error_bit_dq5=1 (memory fail) and     */
/*              and     - read bit is the complement of the bit */
/*                      being programmed or erased              */
/*                                                              */
/*  This function controls the data polling during              */
/*  programming or erasing                                      */
```

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

char CONTROL_DATA_POLLING_FLA(char last_byte,char xhuge *addr)
{
char flag_data_polling=1,data_polling_bit_dq7;
char n_bit,last_bit_q7,error_bit_dq5;
int count7=0;

 last_bit_q7=last_byte&0x80;
 do
 {
 n_bit=7;
 data_polling_bit_dq7 = READ_BIT(n_bit,addr);
 n_bit=5;
 error_bit_dq5=READ_BIT(n_bit,addr);
 count7++;
 }
 while ((data_polling_bit_dq7!=last_bit_q7)&&(error_bit_dq5==0));
 if(error_bit_dq5==0) flag_data_polling=0;
 else
 {
 n_bit=7;
 data_polling_bit_dq7=READ_BIT(n_bit,addr);
 if(data_polling_bit_dq7==last_bit_q7) flag_data_polling=0;
 else RST(addr);
 }
 return(flag_data_polling);
}

/**FUNCTION READ_BIT: TO READ A BIT(x) FROM A FLASH BYTE*********/
/*                                                             */
/*      input variable :        addr, read address             */
/*                              n_bit, bit number              */
/*      output variable :       bit_dqx                        */
/*      this function reads a byte at the given address        */
/*      makes a mask on the byte and returns the value         */
/*      of the bit                                             */
/*************************************************************/

char READ_BIT(char n_bit, char xhuge *addr ) {
char read,bit_dqx;

        read = READ(addr);
        switch(n_bit)
```

```
        {
        case 0: bit_dqx= read&0x01;
                break;
        case 1: bit_dqx= read&0x02;
                break;
        case 2: bit_dqx= read&0x04;
                break;
        case 3: bit_dqx= read&0x08;
                break;
        case 4: bit_dqx= read&0x10;
                break;
        case 5: bit_dqx= read&0x20;
                break;
        case 6: bit_dqx= read&0x40;
                break;
        case 7: bit_dqx= read&0x80;
                break;
        }
        return(bit_dqx);
}

/**FUNCTION RST: TO RESET THE FLASH MEMORY**********************/
/*                                                            */
/*      input variable :        addr, write address           */
/*                              byte=0xf0                     */
/*      output variable :       none                          */
/*                                                            */
/*  This function resets the flash, after a wait state of 5us.  */
/*  Subsequent read operations will read the memory array       */
/*  addressed and output the read byte.                       */
/************************************************************/

void RST(char xhuge *addr) {
char byte,count=0;

 byte=byte_rst;
 WRITE(addr, byte);
 while(count++<100);

}
```

Example of a Program Using the Single Sector Erase and Byte Program Functions

This program fragment erases one sector of Flash memory, and then writes the value 04H throughout the sector using the byte programming operation. It also demonstrates the use of the RESET and CONTROL_DATA_POLLING_FLA functions.

```
/* FLASH+ ; M39432 DRIVER SOURCE CODE : SECTOR ERASE AND BYTE PROGRAMMING   */
/* Version: 1.01                                                            */

/**************************************************************************/
/*         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.                                                   */
/**************************************************************************/

/*****************************************************************/
/*                                                               */
/*  This program calls the functions SECTOR_ERASE and BYTE_WRI_FLA */
/*  input variable :    none                                     */
/*  internal variable : array, which contains the sector numbers  */
/*                      n, number of sector to be erased         */
/*                      erase_ok, flag which indicates if        */
/*                      the erase is ok                          */
/*                      addr_flash_appli, address to program the */
/*                      application in the flash                 */
/*                      byte, byte to program                    */
/*  output variable :  none                                      */
/*  erase_ok=0 when the erase of sectors is successful           */
/*  write_ok=0 when the write in the flash is successful         */
/*****************************************************************/
#define byte1 0xaa
#define byte2 0x55
#define byte_prog 0xa0
#define byte_erase 0x80
#define byte_confirm 0x10
#define byte_sector_erase_confirm 0x30
#define byte_rst 0xf0
#define page 64
#define prog_code1 0x85555
#define prog_code2 0x82aaa
```

```
                        /* DECLARATION OF FUNCTIONS */

                        /* write function*/
 char BYTE_WRITE_FLA(char xhuge *,char);
 void WRITE(char xhuge *,char );
                        /* erase function*/
 char SECTOR_ERASE(char,char []);
                        /* control status register functions */
 char CONTROL_DATA_POLLING_FLA(char,char xhuge *);
 char CONTROL_TIMER(char xhuge *,char);
                        /* read functions */
 char READ_BIT(char, char xhuge * );
 char READ(char xhuge *);
                        /* reset function */
 void RST(char xhuge *);

 char xhuge * SELECT_SECTOR (char);

                       /* MAIN STARTS HERE */

void main(void)
{
char erase_ok=1,array[8],n;
char byte,write_ok_byte=1;
int i=0;

char xhuge *addr_flash_appli;

addr_flash_appli = 0x90000;

/* init array  */
        array[0]=1;
        n=1;
/***************/

        erase_ok = SECTOR_ERASE(n,array);

/* init of variables */

  byte=0x04;
/*********************/

  for(i=0;i<=8192;i++)
```

```
 {
 write_ok_byte=BYTE_WRITE_FLA((addr_flash_appli + i),byte);
 }

}
                        /* END OF MAIN */

/**FUNCTION SECTOR_ERASE : TO ERASE SECTORS OF FLASH MEMORY******/
/*                                                              */
/*   input variable :   n, number of sectors to be erased.      */
/*                      array, contains the sector addresses    */
/*                      (0 to 7) to be erased.                  */
/*                                                              */
/*   internal variables :  erase_timer_bit_dq3, this bit from   */
/*                         the status register gives information*/
/*                         about the erase timeout period. This */
/*                         bit = 0 when the timeout is not      */
/*                         expired.                             */
/*                         flag_timer_dq3, this flag=0 when the */
/*                         internal erase cycle has started.    */
/*                         flag_polling_dq7, receives the result*/
/*                         of the data polling. This flag=0     */
/*                         when the erase is complete           */
/*                         n_sector, sector number              */
/*                         n_bit, bit number                    */
/*                                                              */
/*      output variable :  erase_ok                             */
/*                                                              */
/*  This function issues the sector erase command.              */
/*  All the sectors mentioned in the array are erased.          */
/*  This function returns 0 when there is a successful erase    */
/**************************************************************/

char SECTOR_ERASE(char n,char array[])  {

 char xhuge *addr;
 char byte, flag_polling_dq7=1,erase_timer_bit_dq3
 char n_bit=3,n_sector,i=0,flag_timer_dq3;
 char erase_ok=1, last_byte= 0xff;

/* coded cycle */
 addr= prog_code1;/* write 0xaa at @5555H IN FLASH (base=80000) */
 byte=byte1;
 WRITE(addr,byte);
```

```
 addr= prog_code2;/* write 0x55 at @2AAAH IN FLASH (base=80000) */
 byte=byte2;
 WRITE(addr,byte);

/* erase command */
 addr=prog_code1;/* write 0x80 at @5555H IN FLASH (base=80000)  */
 byte=byte_erase;
 WRITE(addr,byte);

/* coded cycle */
 addr= prog_code1;/* write 0xaa at @5555H IN FLASH (base=80000) */
 byte=byte1;
 WRITE(addr,byte);

 addr= prog_code2;/* write 0x55 at @2AAAH IN FLASH (base=80000) */
 byte=byte2;
 WRITE(addr,byte);
/*sector erase confirm */
do
   {
   n_sector=array[i];
   addr=SELECT_SECTOR(n_sector);    /* write 0x10 at @5555H   */
                                    /* IN FLASH (base=80000)  */
   byte=byte_sector_erase_confirm;
   WRITE(addr,byte);
   erase_timer_bit_dq3=READ_BIT(n_bit,addr);
   i++;
   }
while((i<n)&&(!erase_timer_bit_dq3));

if(!erase_timer_bit_dq3)flag_timer_dq3 = CONTROL_TIMER(addr,n_bit);

flag_polling_dq7= CONTROL_DATA_POLLING_FLA(last_byte,addr);

if((!flag_timer_dq3)&&(!flag_polling_dq7)) erase_ok=0;
return(erase_ok);
}

/**FUNCTION READ: TO READ A BYTE FROM FLASH MEMORY***************/
/*                                                               */
/*      input variable :        addr, read address               */
/*                                                               */
/*      output variable :       read, read byte                  */
```

```
/*                                                                  */
/*  this function reads a byte at the given address                 */
/*******************************************************************/

char READ(char xhuge *addr) {
char read;

      read = *addr;
      return(read);
}

/**FUNCTION CONTROL_DATA_POLLING_FLA (FLASH)*********************/
/*                                                                  */
/*      input variable :        addr, read address                  */
/*                              last_byte, last byte written        */
/*                                                                  */
/*  internal variables :        read bits from the status           */
/*                              register:data_polling_bit_dq7       */
/*                                                                  */
/*  When a programming operation is in progress, this bit           */
/*  outputs the complement of the bit being programmed on dq7.      */
/*  During an Erase operation, it outputs '0' then '1' after        */
/*  Erase completion.                                               */
/*                                                                  */
/*  error_bit_dq5 outputs 1 when there is an error                  */
/*  during programming or erasing.                                  */
/*                                                                  */
/*   last_bit_dq7 variable which stores the value of the last       */
/*   bit dq7 being programmed is set to 0FFH before the erasing.    */
/*   n_bit, bit number to be read.                                  */
/*                                                                  */
/*   output variable :  flag_data_polling, returns 0 when           */
/*                      the read bit = last_bit written             */
/*                      or when dq7=1 for erasure.                  */
/*                      It returns 1 if :                           */
/*                      - error_bit_dq5=1 (memory fail) and         */
/*              and     - read bit is the complement of the bit     */
/*                      being programmed or erased                  */
/*                                                                  */
/*  This function controls the data polling during                  */
/*  programming or erasing                                          */
/*******************************************************************/

char CONTROL_DATA_POLLING_FLA(char last_byte,char xhuge *addr)
```

```
{
char flag_data_polling=1,data_polling_bit_dq7;
char n_bit,last_bit_q7,error_bit_dq5;
int count7=0;

 last_bit_q7=last_byte&0x80;
 do
 {
 n_bit=7;
 data_polling_bit_dq7 = READ_BIT(n_bit,addr);
 n_bit=5;
 error_bit_dq5=READ_BIT(n_bit,addr);
 count7++;
 }
 while ((data_polling_bit_dq7!=last_bit_q7)&&(error_bit_dq5==0));
 if(error_bit_dq5==0) flag_data_polling=0;
 else
 {
 n_bit=7;
 data_polling_bit_dq7=READ_BIT(n_bit,addr);
 if(data_polling_bit_dq7==last_bit_q7) flag_data_polling=0;
 else RST(addr);
 }
 return(flag_data_polling);
}

/**FUNCTION READ_BIT: TO READ A BIT(x) FROM A FLASH BYTE*********/
/*                                                              */
/*      input variable :        addr, read address              */
/*                              n_bit, bit number               */
/*      output variable :       bit_dqx                         */
/*      this function reads a byte at the given address         */
/*      makes a mask on the byte and returns the value          */
/*      of the bit                                              */
/**************************************************************/

char READ_BIT(char n_bit, char xhuge *addr ) {
char read,bit_dqx;

        read = READ(addr);
        switch(n_bit)
        {
        case 0: bit_dqx= read&0x01;
                break;
```

```
        case 1: bit_dqx= read&0x02;
                break;
        case 2: bit_dqx= read&0x04;
                break;
        case 3: bit_dqx= read&0x08;
                break;
        case 4: bit_dqx= read&0x10;
                break;
        case 5: bit_dqx= read&0x20;
                break;
        case 6: bit_dqx= read&0x40;
                break;
        case 7: bit_dqx= read&0x80;
                break;
        }
        return(bit_dqx);
}

/**FUNCTION RST: TO RESET THE FLASH MEMORY**********************/
/*                                                             */
/*      input variable :        addr, write address            */
/*                              byte=0xf0                      */
/*      output variable :       none                           */
/*                                                             */
/*  This function resets the flash, after a wait state of 5æs.  */
/*  Subsequent read operations will read the memory array      */
/*  addressed and output the read byte.                        */
/*************************************************************/

void RST(char xhuge *addr) {
char byte,count=0;

 byte=byte_rst;
 WRITE(addr, byte);
 while(count++<100);

}

/**FUNCTION SELECT_SECTOR***************************************/
/*                                                             */
/*  input variable :  n_secteur, sector to be selected.        */
/*  output variable : addr, base address of the selected sector */
/*                                                             */
/*  This function receives the number of the sector selected   */
```

```
/*  and returns its base address                                    */
/*******************************************************************/

char xhuge * SELECT_SECTOR (char n_secteur) {
char xhuge *addr;
        switch(n_secteur)
          {
        case 0: addr= 0x80000;
                break;
        case 1: addr= 0x90000;
                break;
        case 2: addr= 0xa0000;
                break;
        case 3: addr= 0xb0000;
                break;
        case 4: addr= 0xc0000;
                break;
        case 5: addr= 0xd0000;
                break;
        case 6: addr= 0xe0000;
                break;
        case 7: addr= 0xf0000;
                break;
        }
        return(addr);
}

/**FUNCTION CONTROL_TIMER:(memory flash)************************/
/*                                                             */
/*   input variable : addr,  read address                      */
/*                    n_bit, the number of the bit to be       */
/*                           controlled                        */
/*   output variable : flag_timer, return 0 erase timer bit = 1 */
/*                     return 1 if the count is overflow.      */
/*  This function awaits the start of the internal flash erase  */
/*  cycle. When the last sector erase command has been entered, */
/*  the P/E.C. sets the erase bit timer to '0'.                */
/*  The wait period is finished after 80 to 120us.             */
/*  When the internal erase cycle starts : the erase bit       */
/*  timer = 1 and the function return 0.                       */
/*  If an error occurs (count is overflow the function return 1)*/
/*******************************************************************/

char CONTROL_TIMER(char xhuge *addr,char n_bit) {
```

```
char flag_timer=1,bit_timer;
int count=0;
   do
   {
   bit_timer = READ_BIT(n_bit,addr);
   count++;
   }
   while ((bit_timer != 0x20)&&(count<=100));  /*while dq5 != 1*/
   if(count<=400) flag_timer = 0 ;
   return(flag_timer);
}

/**FUNCTION BYTE_WRITE_FLA: TO WRITE A BYTE IN FLASH & CONTROL***/
/*                                                              */
/*      input variable :        addr, write address             */
/*                              byte                            */
/*      output variable :       flag_data_polling_dq7           */
/*                                                              */
/*  This function writes a byte at the given address and        */
/*  controls the toggle bit.                                    */
/*  return flag_timer_dq6,=0 if byte write is successful        */
/**************************************************************/

char BYTE_WRITE_FLA(char xhuge *addr,char byte)  {
char  flag_data_polling_dq7=1;

/* this block enables the writes in the memory */
       {
char xhuge *address;
char bytes;
address= prog_code1;/*write 0xaa at @5555H IN FLASH (base=80000)*/
bytes=byte1;
WRITE(address,bytes);

address= prog_code2;/*write 0x55 at @2AAAH IN FLASH (base=80000)*/
bytes=byte2;
WRITE(address,bytes);

address=prog_code1;/*write 0xa0 at @5555H IN FLASH (base=80000)*/
bytes=byte_prog;
WRITE(address,bytes);
        }
```

```
        WRITE(addr,byte);
        flag_data_polling_dq7 = CONTROL_DATA_POLLING_FLA(byte,addr);
        return(flag_data_polling_dq7);
}
```

Example of a Program Using the SDP Enable, SDP Disable and SDP Byte Write Functions

This program fragment enables the software data protection function, and then continues to write values into the EEPROM area by using the SDP_BYTE function. Finally, it disables the software data protection function. It also demonstrates the use of the CONTROL_TOGGLE_EE function.

```
/*   FLASH+ ; M39432 DRIVER SOURCE CODE : SDP IN EEPROM BLOCK                */
/* Version: 1.01                                                             */

/*****************************************************************************/
/*         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.                                                     */
/*****************************************************************************/

/*************************************************************/
/*                                                           */
/*      This program calls three functions:                  */
/*      SDP :           enable data protection               */
/*      WRITE_SDP :     write a byte and control the toggle bit */
/*                      when the eeprom is protected         */
/*      DIS_SDP :       disable the data protection          */
/*                                                           */
/*      input variable : none                                */
/*      internal variable :     addr, address to be written  */
/*                              byte, byte to be written     */
/*                              write_ok_byte                */
/*                              free, =0 when the internal   */
/*                              programming cycle is finished */
/*                              (result of the toggle bit)   */
/*      output variable : none                               */
/*                                                           */
/*  write_ok_byte=0 when the write operation is successful   */
/*************************************************************/

#define byte1 0xaa
#define byte2 0x55
#define byte_sdp 0xa0
#define byte3 0x80
#define byte4 0x20
```

```
#define addr1 0x45555
#define addr2 0x42aaa

/* control status register function */
char CONTROL_TOGGLE_EE(char xhuge *);

/* write and read functions */
char READ_BIT(char, char xhuge *);
void WRITE(char xhuge *,char);
char SDP_BYTE(char xhuge *,char);
char READ(char xhuge *);

/* set and disable data protection function */
void SDP(void);
void DIS_SDP(void);

                                    /* MAIN STARTS HERE */
/****************/
/* main program */
/****************/

void main (void)  {

char write_ok_byte=1,free=1 ;
char byte;
char xhuge *addr;

/* INIT THE ADDRESS TO WRITE IN THE EEPROM AND THE DATA   */
addr = 0x40040;
byte = 0xaa;

       SDP();
/* the memory is protected */
       free= CONTROL_TOGGLE_EE(addr);
/* wait the end of the internal write cycle */
       if(free==0);
       {
       write_ok_byte = SDP_BYTE(addr,byte);
       }
       DIS_SDP();

/* the memory is not protected */
```

```
}                                   /* END OF MAIN */

/**FUNCTION WRITE: TO WRITE A BYTE IN EEPROM********************/
/*                                                             */
/*      input variable :        addr, address to be written    */
/*                              byte, byte to be written       */
/*      output variable :       none                           */
/*                                                             */
/* this function writes a byte at the given address            */
/*************************************************************/

void WRITE(char xhuge *addr,char byte)  {

        *addr=byte;
}

/**FUNCTION READ: TO READ A BYTE FROM EEPROM********************/
/*                                                             */
/*      input variable : addr, read address                    */
/*                                                             */
/*      output variable : read, read byte                      */
/*  this function reads a byte at the given address            */
/*************************************************************/

char READ(char xhuge *addr) {
char read;

      read = *addr;
      return(read);
}

/**FUNCTION WRITE_SDP : TO WRITE A BYTE IN EEPROM WITH CONTROL***/
/*                                                             */
/*      input variable :        addr, address to be written    */
/*                              byte, byte to be written       */
/*      output variable :       flag_toggle_dq6                */
/*                                                             */
/*  this function writes a byte at the given address and       */
/*  controls the toggle bit.                                   */
/*  return flag_toggle_dq6,=0 if the byte write is successful  */
/*************************************************************/

char SDP_BYTE(char xhuge *addr,char byte)  {
char  flag_toggle_dq6=1;
```

```
/* this block enables the write in the memory */
       {
char xhuge *address;
char bytes;
    address=addr1;/* write 0xaa at @5555H IN EEPROM (base=40000) */
    bytes=byte1;
    WRITE(address,bytes);

    address=addr2;/* write 0x55 at @2AAAH IN EEPROM (base=40000) */
    bytes=byte2;
    WRITE(address,bytes);

    address=addr1;/* write 0xa0 at @5555H IN EEPROM (base=40000) */
    bytes=byte_sdp;
    WRITE(address,bytes);
        }

    WRITE(addr,byte);
    flag_toggle_dq6 = CONTROL_TOGGLE_EE(addr);
    return(flag_toggle_dq6);
}
/**FUNCTION CONTROL_TOGGLE_EE (EEPROM)**************************/
/*                                                             */
/*      input variable :        addr, read address             */
/*      internal variable :     first_bit_dq6, first read      */
/*                              (from the status register) of  */
/*                              bit dq6                        */
/*                              second_bit_dq6, second read    */
/*                              (       "       ) of bit d6    */
/*                                                             */
/*      output variable :       flag_toggle, returns 0 when the */
/*                              toggle bit stops               */
/*                              (the internal write cycle is   */
/*                              finished) and return 1 if the  */
/*                              count is overflow or less than */
/*                              we expected                    */
/*              this function controls the toggle bit          */
/*************************************************************/

char CONTROL_TOGGLE_EE(char xhuge *addr) {
char first_bit_dq6,second_bit_dq6,flag_toggle=1,n_bit=6;
int count6=0;
        do
```

```
        {
        first_bit_dq6=READ_BIT(n_bit,addr);
        second_bit_dq6=READ_BIT(n_bit,addr);
        count6++;
        }
        while ((first_bit_dq6!=second_bit_dq6)&&(count6<=200));
        if ((count6>=34)&&(count6<=200)) flag_toggle=0;
        return(flag_toggle);
}

/**FUNCTION READ_BIT: TO READ A BIT(x) FROM AN EEPROM BYTE*******/
/*                                                              */
/*      input variable :        addr, read address              */
/*                              n_bit, number of the bit        */
/*      output variable :       bit_dqx                         */
/*      this function reads a byte at the address,              */
/*      makes a mask on the byte and returns the bit value      */
/****************************************************************/

char READ_BIT(char n_bit, char xhuge *addr ) {
char read,bit_dqx;

        read = READ(addr);
        switch(n_bit)
        {
        case 0: bit_dqx= read&0x01;
                break;
        case 1: bit_dqx= read&0x02;
                break;
        case 2: bit_dqx= read&0x04;
                break;
        case 3: bit_dqx= read&0x08;
                break;
        case 4: bit_dqx= read&0x10;
                break;
        case 5: bit_dqx= read&0x20;
                break;
        case 6: bit_dqx= read&0x40;
                break;
        case 7: bit_dqx= read&0x80;
                break;
        }
        return(bit_dqx);
}
```

```
/**FUNCTION SDP: SET SDP(EEPROM)*********************************/
/*                                                               */
/*      input variable :        none                             */
/*      output variable :       none                             */
/*                                                               */
/*       This function enables the data protection               */
/*****************************************************************/

void SDP(void) {
char xhuge *addr;
char byte;

    addr=addr1;/* write 0xaa at @5555H IN EEPROM (base=40000) */
    byte=byte1;
    WRITE(addr,byte);

    addr=addr2;/* write 0x55 at @2AAAH IN EEPROM (base=40000) */
    byte=byte2;
    WRITE(addr,byte);

    addr=addr1;/* write 0xa0 at @5555H IN EEPROM (base=40000) */
    byte=byte_sdp;
    WRITE(addr,byte);

}

/**FUNCTION DIS_SDP:  DISABLE SDP(EEPROM)***********************/
/*                                                               */
/*      input variable :        none                             */
/*      output variable :       none                             */
/*                                                               */
/*        This function disables the data protection             */
/*****************************************************************/

/* DISABLE SDP */

void DIS_SDP(void) {
char xhuge *addr;
char byte;

    addr=addr1;/* write 0xaa at @5555H IN EEPROM (base=40000) */
    byte=byte1;
    WRITE(addr,byte);
```

```
    addr=addr2;/* write 0x55 at @2AAAH IN EEPROM (base=40000) */
    byte=byte2;
    WRITE(addr,byte);

    addr=addr1;/* write 0x80 at @5555H IN EEPROM (base=40000) */
    byte=byte3;
    WRITE(addr,byte);

    addr=addr1;/* write 0xaa at @5555H IN EEPROM (base=40000) */
    byte=byte1;
    WRITE(addr,byte);

    addr=addr2;/* write 0x55 at @2AAAH IN EEPROM (base=40000) */
    byte=byte2;
    WRITE(addr,byte);

    addr=addr1;
    byte=byte4;/* write 0x20 at @5555H IN EEPROM (base=40000) */
    WRITE(addr,byte);
}
```

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

apps.eeprom@st.com

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