# How to Erase and Write the EEPROM in the 9S08DZ MCU

By: Ake Webjorn
Rafael Peralez
Technical Information Center

**Table Of Contents**

**How to Eras1e/Write the EEPROM in ths S08DZ MCU**

## 1 Introduction

This paper describes how to use the non-volatile EEPROM memory in the MC9S08DZ Microcontrollers. Its focus is to give a brief explanation on the non-volatile EEPROM and explain the software needed in order to erase and write the EEPROM. It does not describe the FLASH erasure/programming. It is quite similar though.

## 2 The non-volatile EEPROM

The memory in the MC9S08DZ consists of RAM and two kinds of non-volatile memory, the FLASH and EEPROM. All the circuitry to generate the high voltage required to program/erase the FLASH and EEPROM is part of the device; this avoids the need of any external circuitry or voltage (besides the supply voltage) to either erase or program the FLASH or the EEPROM. During erasure/programming of the FLASH or EEPROM it is not possible to execute instructions from within the memory block which is being erased/programmed; this means that to program the FLASH, the code must be either in RAM or in EEPROM (unless the device has more than one FLASH block like the S08LC60 or S08QE128 for example), to program the EEPROM, the code should be executed either from FLASH or from RAM.

Since this document is intended to show the way to work with the EEPROM, all the routines will be stored into FLASH. In case that the user wants to use the same code to also program and erase the FLASH it will be necessary to copy these routines into RAM and take care of the differences in the sector sizes and memory protection between FLASH and EEPROM.

The EEPROM/FLASH memory varies from 512 bytes/16 K byte for the MC9S08DZ16 up to 2048 bytes/60 K bytes for the MC9S08DZ60.

## 2.1 EEPROM Memory Mapping

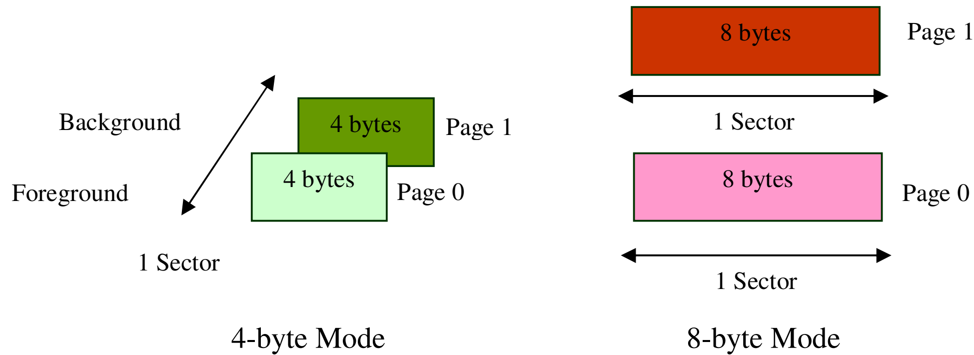



Figure 2.1 Two ways of dividing the EEPROM

The EEPROM is divided into Sectors, where each Sector consists of 8- bytes. As shown in **Figure 2.1.1,** there are two ways of configuring the EEPROM memory map.

- 4 Byte Mode: This mode is selected when the EPGMOD bit in the FOPT register is equal to 0. In this mode, each 8-byte sector slits four bytes in foreground and four bytes on background on the same addresses. The EPGSEL bit in the FLASH and EEPROM Configuration register (FCNFG) selects which page is active and determines which 4-bytes can be accessed at any given time. When an erase operation is performed the 8-byte sector (4-bytes on foreground and 4-bytes in background) will be erased.
- 8 Byte Mode: When the EPGMOD is set to 1, all 8 bytes of a sector are available in the same page. When changing the EPGSEL bit a full 8-byte sector in the foreground is swapped with its equivalent background sector. When performing an erase operation, the 8-byte sector in foreground will be erased, while the same locations in background won't be changed.

As the NVOPT register is in FLASH, this value cannot be changed dynamically. The NVOPT register is automatically copied into the read-only FOPT register at power on. In this register is the EPGMOD bit which selects between 4-byte or 8-byte mode so the mode is something that can not be changed during run time.

The Pages are mapped as a foreground Page and a background Page. The foreground page is the only one that can be accessed while the background Page is not accessible at all. The EPGSEL bit selects which page is foreground and which is background, giving access to the whole memory map; this bit can be changed during run time. The Sector is the smallest part of the EEPROM that can be erased; in the case of EEPROM is 8-bytes for both modes; the erased byte contains 0xFF (all bits set to one).

**Note that when the Sector is erased in 4-bytes mode, both the foreground and the background Pages are erased.**

Programming the EEPROM can be made in a byte-wise manner. Only an erased byte should be programmed and only the foreground bytes can be read or programmed.

There is a burst programming method that can be used to program sequential bytes of data at about twice the speed of the normal programming method. In the burst mode up to four Sectors or 32 sequential bytes may be programmed at once. The first byte in the sequence will take the same amount of time to program as a byte programmed using the standard mode. Subsequent bytes will be programmed faster. When going outside this 32-byte boundary, the next byte will take the normal programming time

## 2.2 Program and Erase Times

For programming and erasing operations, the FLASH and the EEPROM timings are based on their own clock. In the S08 family, the FLASH clock is derived from the bus clock divided by the contents of the FLASH Clock Register, FCLK. The division of the Bus Clock by the FCLK register must set the internal flash clock to a frequency ( $f_{FCLK}$ ) between 150 kHz and 200 kHz. This register can be written only once, so normally this write is done during reset initialization. One period of the resulting clock (1/ $f_{FCLK}$ ) is used by the command processor to time program and erase pulses. An integer number of these timing pulses are used by the command processor to complete a program or erase command. Both, the FLASH and EEPROM timings are based on the same clock, therefore, writing the FCKL register properly ensures the proper usage of FLASH and EPROM for program and erase operations.

The FCDIV register has 6 bits that are the value that will divide the bus frequency; additionally, the bit PRDIV8 of this register will divide the bus clock 8 times. This bit is used for bus frequencies above 12.8 MHz. The following table shows the appropriate values for the FCDIV register:

| $f_{Bus}$ | PRDIV8 (Binary) | DIV (Decimal) | $f_{FCLK}$ | Program/Erase Timing Pulse (5 µs Min, 6.7 µs Max) |
|---|---|---|---|---|
| 20 MHz | 1 | 12 | 192.3 kHz | 5.2 µs |
| 10 MHz | 0 | 49 | 200 kHz | 5 µs |
| 8 MHz | 0 | 39 | 200 kHz | 5 µs |
| 4 MHz | 0 | 19 | 200 kHz | 5 µs |
| 2 MHz | 0 | 9 | 200 kHz | 5 µs |
| 1 MHz | 0 | 4 | 200 kHz | 5 µs |
| 200 kHz | 0 | 0 | 200 kHz | 5 µs |
| 150 kHz | 0 | 0 | 150 kHz | 6.7 µs |

Figure 2.2 Appropriate values for the FCDIV Register

The time used to perform each operation is an integer number of the flash timing pulses. The allowed commands are Byte Program, Burst Program, Sector Erase, Mass Erase and Sector Erase Abort. The times that each command takes are:

| Parameter | Cycles of FCLK | Time if FCLK = 200 kHz |
|---|---|---|
| Byte program | 9 | 45 µs |
| Burst program | 4 | 20 µs[1] |
| Sector erase | 4000 | 20 ms |
| Mass erase | 20,000 | 100 ms |
| Sector erase abort | 4 | 20 µs[1] |

[1] Excluding start/end overhead

Figure 2.3 FLASH and EEPROM Commands Times

An important thing to notice is that the Burst Command functionality applies only for Flash; in the case of EEPROM it is possible to launch a burst command but it will take the same time that a single write command because of the way EEPROM memory is built.

If the FLASH clock is generating a frequency of 200 kHz (which is the maximum value for the flash clock), then it takes about 20 ms to erase one sector of EEPROM and about 45 us to program a single byte. The mass erase takes about 100 ms. Burst programming is available for Flash and EEPROM but because of the way the EEPROM memory is aligned it takes the same amount to byte write or burst write a byte in EEPROM while is about twice as fast as the normal programming mode.

**Important Note**

Programming or erasing the FLASH or EEPROM with an input clock less than 150 kHz should be avoided; setting such value can destroy the flash due to overstress. Setting a value that results in a flash clock greater than 200 kHz can result in incomplete programming or erasure of the memory array cells.

## 2.3 Comparison with the FLASH

The difference between the Flash EPROM and EEPROM in MC9S08DZ MCU is that the EEPROM is divided into foreground and background pages where only half the memory is accessible at a time, whilst the Flash EPROM is not divided into pages. The EEPROM sector size is 8 bytes, whilst the Flash EPROM sector size is 768 bytes. The EPROM can be used in 4 Byte Mode or 8 Byte Mode. The Flash EPROM does not have this feature.

In both the EEPROM and the FLASH, it is possible to change a single byte at a time. The programming in both cases is done by writing 0 into the bits that should be changed. Reprogramming bits in an already programmed byte without first performing an erase operation may disturb data stored in the EEPROM or FLASH memory.

Another difference is the Burst command functionality; as mentioned before, even when it is possible to use Burst command for sequential writing in EEPROM, it doesn't save time like it does in Flash. This is because the EEPROM array is built in a way that there is only one byte per row in the array so it basically takes the same time to make a byte write than a burst write.

***Note that there is no software difference when erasing the FLASH compared with erasing the EEPROM. The only difference is the size of the erased Sector. In the EEPROM it is bytes, while in the FLASH, it is 768 bytes.***

# 3 How to erase/program the EEPROM

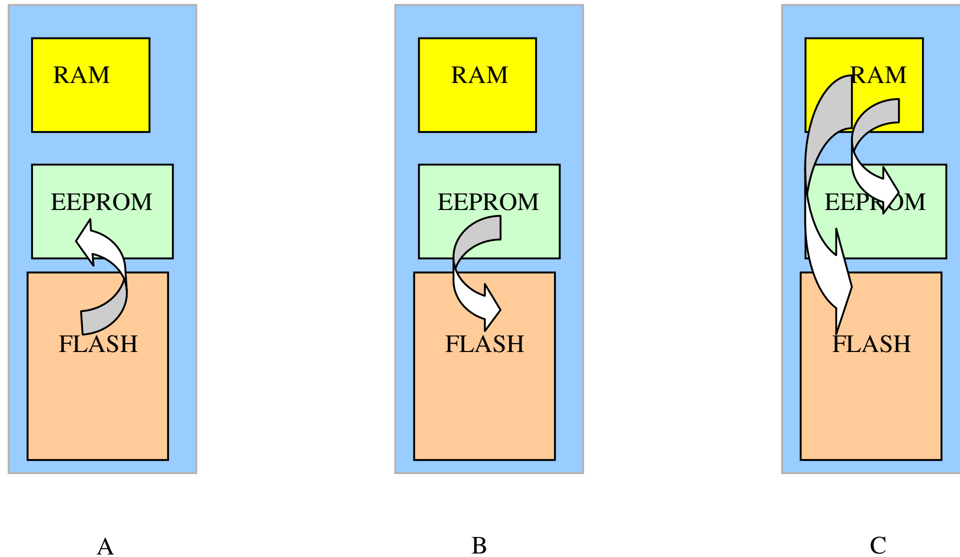



Figure 3.1 Three different ways for programming the EEPROM/FLASH on the 9S08DZ

The address and the data-buses must be kept stable during the time the EEPROM is programmed. This can be done in three ways on the 9S08DZ. Look at **Figure 3.1**. The EEPROM can be programmed as in picture A, the FLASH can be programmed as in figure B or both the EPROM and FLASH can be programmed as in figure C.

The more complex solution in figure C can be used for both, EEPROM and FLASH erasure/programming. The B solution is for programming FLASH and version A is used for EEPROM programming only. This paper will concentrate on the EEPROM solution only, that is the A picture.

## 3.1 Erasing a Sector

As mentioned earlier, there is hardware support that handles the erasure/programming of the EEPROM/Flash EPROM memory. This hardware automatically distinguishes between an EEPROM sector and a Flash EPROM sector and accounts for the difference in the sector size. All that is required is an address which identifies the specific sector to be erased

In the software example provided with this application note a 4- or 8-byte EEPROM Page is erased. As explained in section 2.1 of this document, 4-byte or 8-byte mode, is selected by the EPGMOD bit in the FOPT register.

An address to the Sector should be supplied to the erase subroutine. This address can be any address within the sector to be erased. The erase routine first needs to check that no earlier errors have occurred and the writes anything into the address supplied. This write operation buffers the address. The value of the data written is unimportant as an EEPROM sector is being erased. Then the erase command (0x40) is written into the Flash Command register (FCMD) followed by a write to the FCBEF register which starts the command. The CPU should then read the status register FSTAT to check that everything is OK and the FCCF bit is set indicating the command has finished.

### 3.2 EPROM programming

The steps previously described to erase a sector are the same when performing a byte program operation; the only difference is that whilst the data provided is irrelevant for the erase command while the program operation uses this data to write into non-volatile memory and the FCMD register needs to be written with a Program command (0x20)

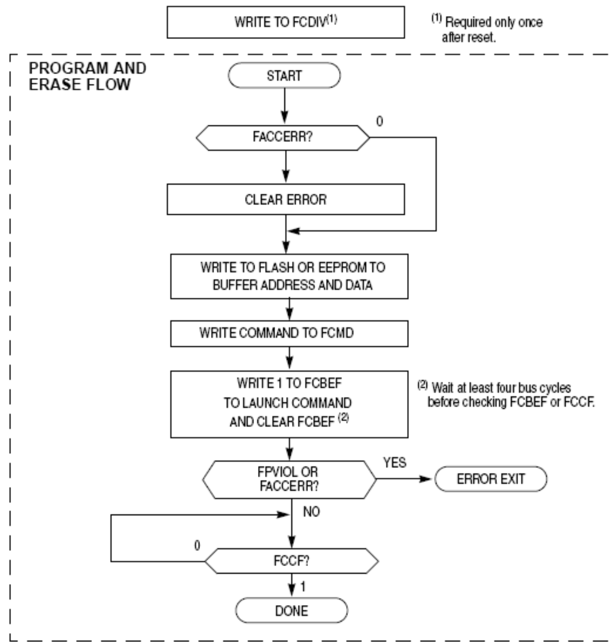



Figure 3.2 Erase and Programming flow chart

## 3.3 Burst Programming

The burst programming is used when a sequential write is needed; even when Burst command doesn't save time when used for EEPROM as it does with Flash programming, the sequence will be explained since it is very similar to a single byte programming and an erase sequence and this will be used later as the base for the Software example. Figure 3.3 shows a Burst Command flow chart:

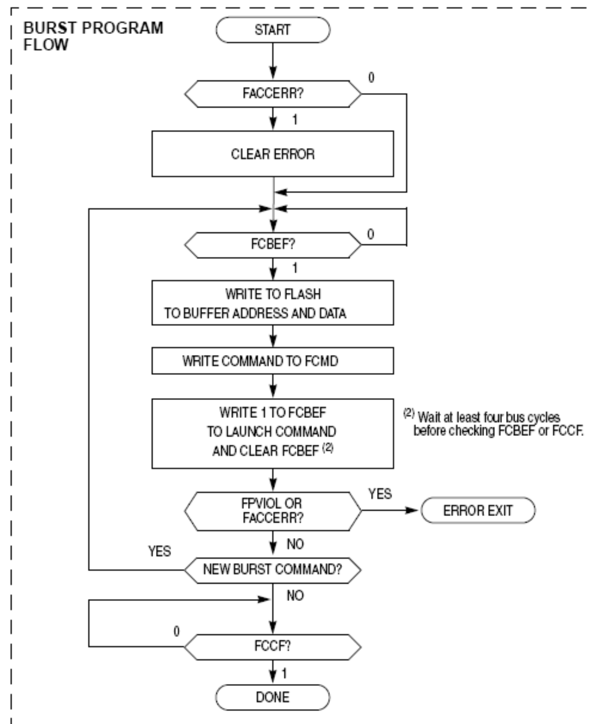



Figure 3.3 Burst Program Flow

When performing a burst command, the user writes the address and data and then writes a 25 hexadecimal to the FCMD register to indicate that a burst command will be performed; after this the command is launched and the error flags are checked and, in case that a new command is going to be performed, then you just need to wait for the FCBEF to start the new command execution. When the last byte is written, then the code must wait for the FCCF flag.

## 3.4 Additional Non-volatile Memory Protection

The FLASH should be protected by using special protection registers NVPROT and FPROT. NVPROT lies at the top of the FLASH memory, located at address 0xFFBD. After reset, the contents of register NVPROT are copied into FPROT; FPROT is readable all the time and writable only to increase the amount of protected memory. Any write to FPROT that tries to decrease the amount of protected memory will be ignored.

The value written into FPROT specifies the highest memory address that is write and erase protected. The two EPS bits of this register are available for the EEPROM protection which offers 0, 32, 64 or 128 protected bytes. The following table shows the possible combinations for this section of the FPROT register:

| EPS | Address Area Protected | Memory Size Protected (bytes) | Number of Sectors Protected |
|---|---|---|---|
| 0x3 | N/A | 0 | 0 |
| 0x2 | 0x17F0 - 0x17FF | 32 | 4 |
| 0x1 | 0x17E0 - 0x17FF | 64 | 8 |
| 0x0 | 0x17C0–0x17FF | 128 | 16 |

Figure 3.4 EEPROM Protection Combinations

# 4 The Software used

The following figure shows the software architecture used to implement the EEPROM programming and erasing routines:

As mentioned before, due to the similarity between the program, erase and burst commands, a single routine can be used to perform these three flash operations. Figure 4.1 shows the flow charts for the program, erase and burst command operations

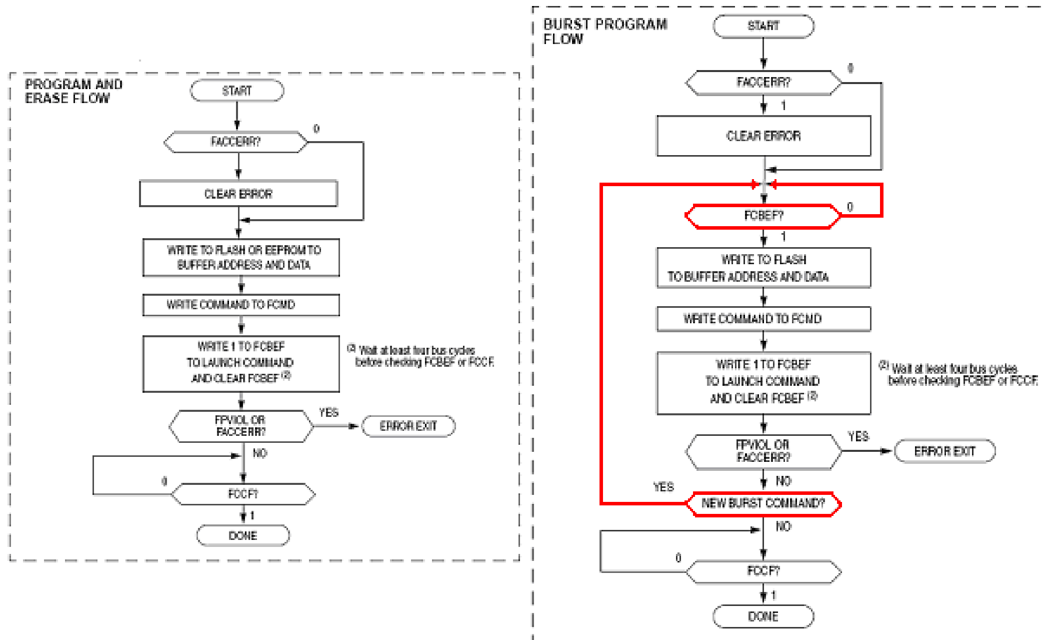
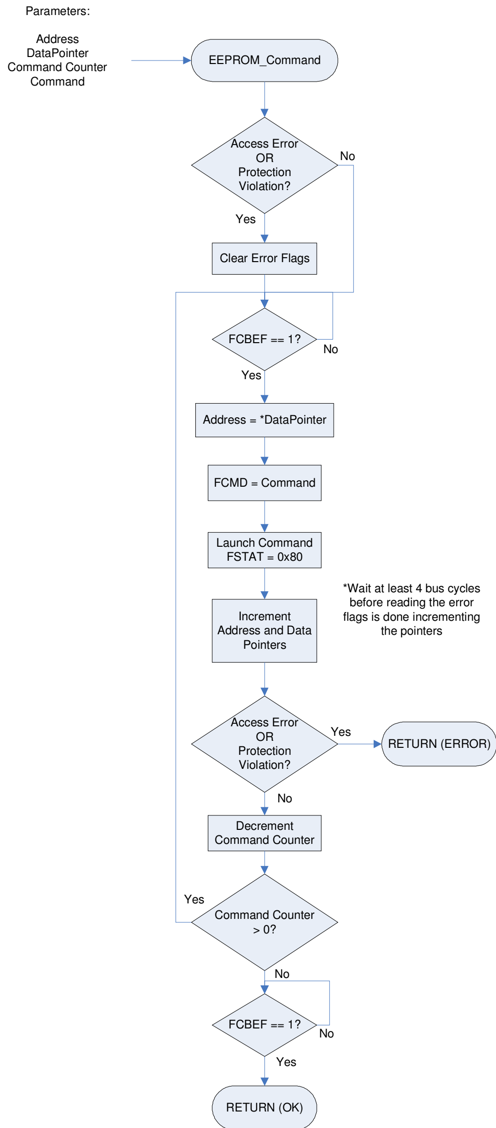



Figure 4.1 Program, Erase and Burst commands Flow Charts

As shown in this figure, there are only two differences (besides the command number) between the Burst command compared with the program and erase commands:

- The first one is that the burst command first needs to wait for the FCBEF; this flag indicates that the command buffer is empty and that a new command sequence can be started
- After the current command has finished and that there was no Protection Violation or Access Error, the code needs to check if there is another command waiting to be executed.

The implementation of a single routine that performs these three operations can be done in the following way:




The routine will receive the following parameters:

- Address is the value in 16-bits where the data is going to be stored
- Data Pointer is the reference where the information to be written is located. It is important to remember that, for the erase command, this can be anything.
- Command Counter is the amount of times that a command will be performed. In this code this makes sense for the burst programming only.
- Command is a parameter that indicates the command that will be written into the FCMD register. This can be a Byte Program, Sector erase or Burst programming command.

For a Sector Erase command the user just needs to provide the address that he wishes to erase; the whole 8-byte sector will be erased after doing this. It depends on either 4-byte or 8-byte mode on what the user will see after performing the erase operation: either 8-byte in foreground will change their values to 0xFF (erased state) or 4-bytes in foreground and 4-bytes in background will change to the erased state.

In the case of a single byte program, the routine should receive the address and the pointer where the byte to be written is; the command counter should be set to 1 for this case since this is a single byte programming. The burst programming is a similar case, the command written to the FCMD register changes to a burst command (25 hexadecimal) and the command counter should have the number of bytes that will be saved in EEPROM.

## 4.1 *The Application Programming Interface (API)*

In the previous pages this document explains the different modes to map the EEPROM, its pages, the programming, erase and burst sequences. The software files provided with this application note are build in a way that a single routine is used to perform the three operation; however, it provides a way to call each of the operations in a different way. Section 5 of this document explains the Software and Hardware requirements and what does the example software does.

All the programming and erasing routines return an Error identifier; if the routine returns EEPROM_OK means that all the operations were performed without problems. In case that any routine returns EEPROM_FAIL it means that the executed command generated either an Access Error or a Protection Violation Error. For more information on what could generate these errors please refer to the Access Errors and Block protection sections of the data sheet.

The available routines for this application note are:

### 4.1.1 Initializing the FLASH and EEPROM Clock

As previously described in this document, the FLASH and EEPROM modules have their own clock which is derived from the bus clock divided by the contents of the FLASH Clock Divider register (FCDIV). It is necessary that the bus clock divided by the contents of this register result in a value between 150 kHz and 200 kHz. To minimize the possibility of error, the DZ_EEPROM.h file calculates the right FCDIV value based on a bus clock definition. The user needs to look for the BUS_CLOCK definition inside this file and modify it to match the bus clock that he will be using.

The following routine must be called before using any erasing or programming routine:

**EEPROMInit(*void*)**

This function clears the error flags in case they are set and writes the right value to the FCDIV register. If the FCDIV register is not initialized the rest of the software provided will not work properly. It is strongly recommended that the user only modifies the BUS_CLOCK definition and calls this function to initialize the FLASH and EEPROM clock divider.

### 4.1.2 Erase a Sector

The following routine is provided to erase the contents of a sector:
**ErrorID = EEPROM_Erase_Sector (*Address*)**
Where *Address* is the location that is going to be erased. The address has to be provided in a 16-bit value. The following things have to be considered when using this routine:

- The 8-byte sector is going to be erased; however, depending on the selected mode (4-byte or 8-byte) is the result of this operation. When in 4-byte mode, the contents of the foreground and background pages are erased while in 8-byte mode only the foreground page is erased.
- The address provided must be a section of EEPROM memory. If the address provided is a FLASH section, then the CPU won't be able to fetch the next instruction from FLASH and the MCU will reset.

When calling this definition, the command is automatically set to a Sector Erase Command, the data is a dummy pointer to 0 and the amount of commands to be executed is set to 1. With this we are going to execute a single erase command in the address provided.

### 4.1.3 Program a Byte, a Word or a Double Word

There is a specific routine that can be called depending if the user wants to write a byte, a word or a double-word in EEPROM. These are:

**ErrorID = EEPROM_Byte_Program(*Address, Byte*)**
**ErrorID = EEPROM_Word_Program(*Address, Word*)**
**ErrorID = EEPROM_DoubleWord_Program(*Address, DoubleWord*)**

For these three functions the format is the same; the user has to provide the address and the data to be programmed. Depending if it's whether a byte, a word or a double word program is the specific behaviour and what the definition is going to set as the parameters. For a byte program, the Address and the data are sent as parameters and then the routine sets the command for a byte program command and the amount of commands to 1.

In the case of a Word command the logic is the same; the difference is that the amount of commands is set to two since the programming routine can only program a single byte per command execution and, since this is a sequence of bytes that will be programmed, the flash command is set to a Burst command instead of a Byte program command. The case of the Double Word program is similar to the Word program, the only difference is that the amount of bytes that will be programmed in EEPROM is 4.

When using the Word and Double Word program it is important to mention that the routine doesn't validate that the data type is either a 16-bit or 32-bit value; it simply takes either 2 or 4 bytes and programs that data into EEPROM. It doesn't validate that the address supplied is part of the EEPROM section; in case that a flash location is provided as a parameter the MCU will reset.

### 4.1.4 Program a Buffer of Bytes

This case can be used to program from 1 to 255 bytes into EEPROM; the 255 bytes limitation is due to the usage of an 8-bit variable to receive the counter. The user can change this for a 16-bit value and should be able to program up to the available 2K bytes EEPROM space in a single command. The available call for this is:

**ErrorID = EEPROM_Buffer_Program(*Address, Size, DataPtr*)**

For this function the user needs to provide the Address where data will be written, the amount of bytes that will be stored in EEPROM and the pointer where the buffer that will be copied into EEPROM is located.

**Important Note**

Before programming a particular byte in the EEPROM, the address that will be programmed needs to be in the erased state (all bits set to 1). Reprogramming bits that are already in a programmed state may disturb the data stored in the EEPROM. The routines provided to write a byte, word, double-word or a buffer assume that the addresses that will be written are already erased and don't erase the sector or perform any kind of validation before writing the data.

### 4.1.5 Using the Pages

Section 2.1 of this document gives a description on the different modes and how to change the active page. Basically, the EPGSEL bit selects which is the page that is mapped as foreground page and which remains as the background page. To change the active page the software provided has the following definitions:

**SET_EEPROM_PAGE0**
**SET_EEPROM_PAGE1**

These are very simple macros that only change the EPGSEL bit and by doing this change the active page. It is important to remember that reading and programming operations can be only performed in the active page; if the user sets the page to 0, all the data read and all the programming operations will be done in this page. For erasing operations it depends on the active mode (either 4-bytes or 8-bytes) to determine what is going to happen:

- In 4-bytes mode the foreground and the background page are erased; this means that 4-bytes of the active page are going to change their value to 0xFF and 4-bytes in the exact same location but in the background page are going to do the same
- For 8-bytes mode, only the contents of the foreground page will be erased; if the user changes the page he will be able to see the contents of the other page

### 4.1.6 Selecting 4-bytes or 8-bytes mode

The software provided doesn’t take care of doing anything automatically for either 4-bytes or 8-bytes mode; the reason is that this byte also takes care of the FLASH security; since this is an important feature, the SW files provided only contain the declaration that controls the mode; this is at the top of the DZ_EEPROM.c file and is implemented in the following way:

```
const unsigned char NVOPT @0xFFBF = 0xFE;
```

According to the bits definition in the datasheet for the NVOPT register a value of 0xFE will set the EEPROM to 8-bytes mode while a value of 0xDE will set the EEPROM to 4-bytes mode.

## 4.2 Porting the code to another application

The software files are designed in a way that they can be ported to other applications without any changes providing the functionality described in this section. In case that the user needs to integrate this code into a different application the following steps need to be followed:

1. Add the DZ_EEPROM .c and .h file in the new project; it is recommended to create a copy of these files into the new project’s directory so in case that a modification is needed, the user can modify only the files of his project without changing the original version
2. Open the DZ_EEPROM.h file and modify the BUS_CLOCK definition according to the Bus clock used in his application. This is a critical step since the FLASH clock divider is calculated automatically at compilation time and is based on this BUS_CLOCK definition. In case that a wrong value is provided here the FLASH and EEPROM clock will be set erroneously and there is a risk of damaging the EEPROM module
3. Include the DZ_EEPROM.h file in any source file where the provided API need to be used; this is needed so any other file know about the definitions and functions existing in the DZ_EEPROM.c file.

With these three simple steps anyone should be able to add the EEPROM routines into a project that uses the S08DZ family.

# 5 The Example Software

The software provided with this application note is a very simple demonstration of the available API; since the focus of this application note is to provide routines that allow a simple way to program and erase the EEPROM the example code is developed in a way that only shows how to do this.

The code was developed and tested in CodeWarrior version 6.1; to be able to test the results described in this section the user needs to install this software and have a programming tool that is able to communicate with the Real Time Debugger that is part of CodeWarrior.

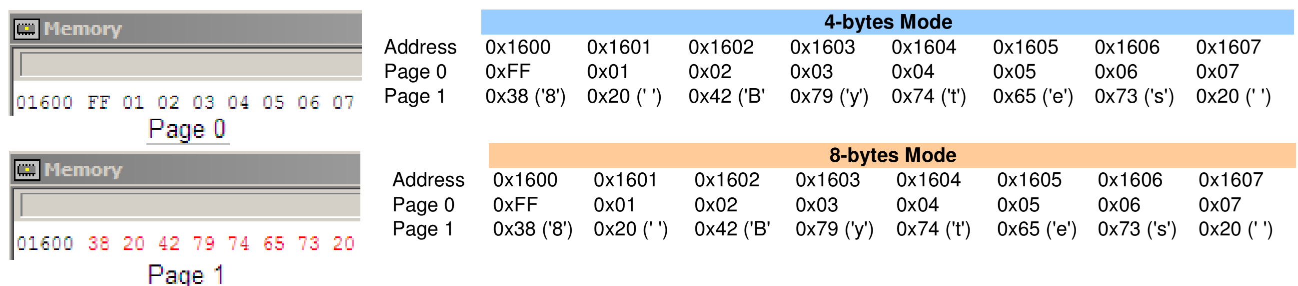
In order to avoid the dependence from external components, the Software provided is not configuring the bus clock from any external crystal or oscillator. The configuration used is the one that is available after reset which provides a 4 MHz bus clock. However, the SW does write the Trim registers from non-volatile addresses 0xFFAE and 0xFFAF with the intention of adjusting the internal clock and provide a more accurate 4 MHz bus clock.

## 5.1 Software Demonstration

After initializing the Trim registers and the FLASH and EEPROM clock divider the software is performing the following operations[1]:

```
unsigned char Buffer[] = {"8 Bytes "};
unsigned char Byte_To_Be_Programmed = 0x01;
unsigned int  Word_To_Be_Programmed = 0x0203;
unsigned long DWord_To_Be_Programmed= 0x04050607;

SET_EEPROM_PAGE0;

EEPROM_Byte_Program(0x1601, Byte_To_Be_Programmed);
EPROM_Word_Program(0x1602, Word_To_Be_Programmed);
EEPROM_DoubleWord_Program(0x1604, DWord_To_Be_Programmed);

SET_EEPROM_PAGE1;

EEPROM_Buffer_Program(0x1600, (sizeof(Buffer)-1), Buffer));
```

At this point the memory map should look in the following way




| | 4-bytes Mode | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 |
| Page 1 | 0x38 ('8') | 0x20 (' ') | 0x42 ('B' | 0x79 ('y') | 0x74 ('t') | 0x65 ('e') | 0x73 ('s') | 0x20 (' ') |

| | 8-bytes Mode | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 |
| Page 1 | 0x38 ('8') | 0x20 (' ') | 0x42 ('B' | 0x79 ('y') | 0x74 ('t') | 0x65 ('e') | 0x73 ('s') | 0x20 (' ') |

Figure 5.1 Memory Map after the write instructions

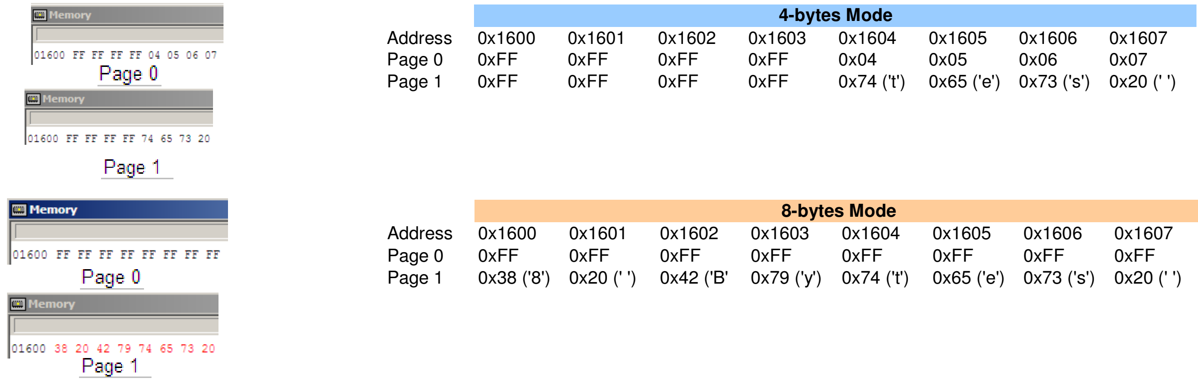
As shown in Figure 5.1, there are no differences in the memory map in either 4-bytes or 8-bytes mode (follow the steps described in section 5.2 of this document to see the right data in

[1] This is not the exact code provided. This section is only intended to describe the functionality provided with this application note

the debugger memory map). The difference will now be shown as the erase operations are performed; the following instructions take place next in the code:

```
SET_EEPROM_PAGE0;

EEPROM_Erase_Sector(0x1600);
```

After these two instructions this is how the memory map looks in the following way:




| 4-bytes Mode | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0xFF | 0xFF | 0xFF | 0x04 | 0x05 | 0x06 | 0x07 |
| Page 1 | 0xFF | 0xFF | 0xFF | 0xFF | 0x74 ('t') | 0x65 ('e') | 0x73 ('s') | 0x20 (' ') |

| 8-bytes Mode | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |
| Page 1 | 0x38 ('8') | 0x20 (' ') | 0x42 ('B' | 0x79 ('y') | 0x74 ('t') | 0x65 ('e') | 0x73 ('s') | 0x20 (' ') |

Figure 5.2 Memory map differences after erasing a sector, 4-bytes vs. 8-bytes Mode

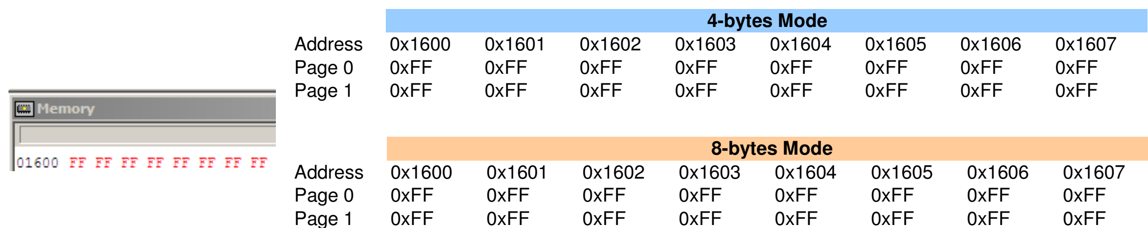
In the previous picture we can see the difference when using 4-bytes and 8-bytes modes;

- When working with 4-bytes mode, after erasing a sector, we'll see that only 4-bytes in the memory map are erased but we also know that other 4-bytes in the background page are erased. This means that 4-bytes of the memory map will be erased for both pages which give us the sector size of 8-bytes.
- For 8-bytes mode we'll see that 8-bytes are erased after performing the sector erase operation; however, the 8-bytes in the same location but in the background page are not affected by this sector erase command. Whenever we change the page we'll see that the contents of the page that was the background page at the time of executing the sector erase command are unaffected.

The routine ends with the following lines:

```
SET_EEPROM_PAGE1;

EEPROM_Erase_Sector(0x1604);
```

With these two lines we change the page and we erase the sector that contains address 0x1604. Even when the behaviour is different for 4-bytes and 8-bytes mode, the result will be that both pages will be erased from address 0x1600 to address 0x1607




| 4-bytes Mode | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |
| Page 1 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |

| 8-bytes Mode | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0x1600 | 0x1601 | 0x1602 | 0x1603 | 0x1604 | 0x1605 | 0x1606 | 0x1607 |
| Page 0 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |
| Page 1 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |

Figure 5.3 Memory Map after erasing the other page

As a good exercise, in case that the erase command is changed in this line for any address between 0x1600 to 0x1603, in 4-bytes mode the contents of addresses 0x1604 to 0x1607 will not be changed while in 8-bytes mode all the contents will be erased.

## 5.2 Tips Working with CodeWarrior

When working with the CodeWarrior debugger there are a couple of things that can make a big difference while debugging the code. The first one is the 4-bytes and 8-bytes mode; this is particularly important because the debugger needs to know which the selected mode is in order to properly display the contents of the memory map and also to be able to initialize any data that we've placed in EEPROM section. To configure the memory map in the debugger follow these steps:

1. Once the debugger is open, click on the Multilink/Cyclone Pro menu
2. Click on Advanced Programming/Debug Options
3. In the opening window select the 4-bytes or 8-bytes mode as shown in the following picture

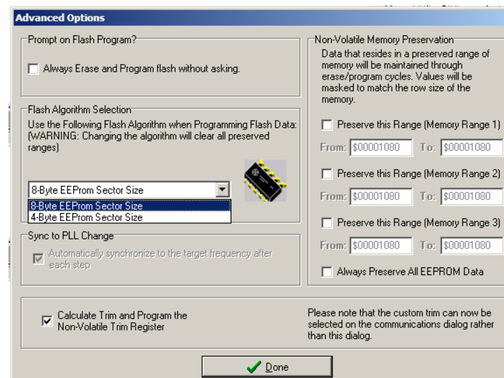



Figure 5.4 Advanced Options for programming and debugging

It is very important that the mode selected matches the value that we will write in the EPGMOD bit the next time we program the part. If the EPGMOD doesn't match the value configured in the debugger it is very likely that we see an erratic behaviour while debugging.

The other tip is due to the fact that EEPROM memory might not be refreshed in the memory window all the time. Specially when changing pages, it is possible that the contents are not refreshed in the window. This doesn't mean that the values are not there, it's simply that they are not being shown by the debugger. To refresh the contents of the memory window:

1. In the debugger click on Multilink/Cyclone Pro menu
2. Click on the Debugging Memory Map option
3. Select the EEPROM addresses range (from 0x1400 to 0x17FF for the DZ60) and click on Modify/Details
4. In the opening window select the checkbox to refresh memory when halting

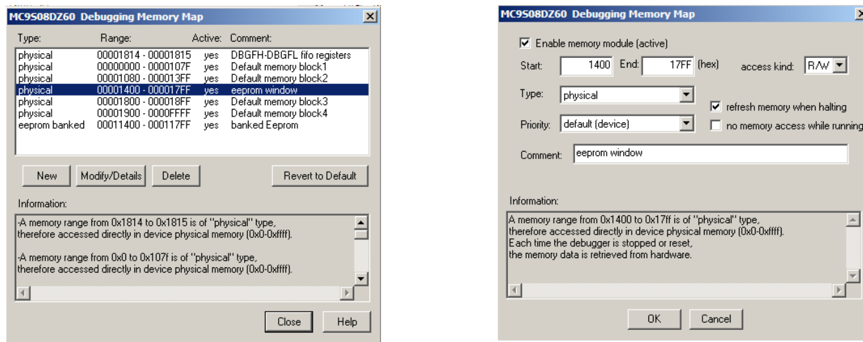



Figure 5.5 Refreshing the memory map when halting

By doing this you should be able to see what's in the memory map at that time. It might be required to do this more than once in a debugging session.