Application Note

AN2302/D
Rev. 1, 1/2003

EEPROM Emulation for the MC9S12C32

MOTOROLA intelligence everywhere™ | digital dna™

By: Stuart Robb
Motorola, East Kilbride

Introduction

Many applications require EEPROM (Electrically Erasable Programmable Read-Only Memory) for non-volatile data storage. EEPROM is typically characterised by the ability to erase and write individual bytes of memory many times over, with programmed locations retaining their data over an extended period when the power supply is removed.

Most MC9S12 Flash microcontrollers include on-chip EEPROM but some omit EEPROM for reduced price. MC9S12 microcontrollers which do not have on-chip EEPROM, such as the MC9S12C family, can store non-volatile data in the on-chip Flash memory using the software described in this application note, thus saving the cost of an external EEPROM.

This application note describes a software method for emulating EEPROM using the on-chip Flash memory of the MC9S12C32. This method may be used on other MC9S12 flash microcontrollers that do not have on-chip EEPROM and may also be used to provide additional EEPROM data storage on MC9S12 flash microcontrollers that have insufficient EEPROM for a particular application. This software makes no assumptions about the way in which the non-volatile data will be stored or updated. Data variables may be of arbitrary size and may be updated at random. It is possible that a more optimised approach may exist for a less general case.

The software described in this application note may be configured to allow interrupts to be serviced during the programming and erasure of non-volatile data. Alternatively, a "call-back" function may be enabled, allowing inputs to be polled or a watchdog to be refreshed during programming and erasure.

EEPROM Emulation software features:

- User configured emulated EEPROM size
- Non-volatile data variables stored in Flash
- ReadEeprom and WriteEeprom functions to access variables
- Low RAM requirements - 35 bytes minimum (plus stack)
- Low Flash requirements - 1024 bytes minimum for emulated EEPROM, 691 bytes minimum for code
- User defined callback function during program/erase
- Interrupt servicing during program/erase is possible

Implementation

There are two common methods used to store non-volatile data in Flash. One method is to keep a copy of all non-volatile data variables in a buffer in RAM. The data is "saved" to Flash as often as is considered necessary by programming the entire buffer contents to a pre-erased Flash sector. This method is relatively simple to implement, permits the data variables to be read at all times and allows control of the number of program/erase cycles. The major disadvantage is that a large amount of RAM is required, as both the data buffer and the programming routine must be located in RAM. There is also a risk of losing data if a reset should occur after updating the RAM buffer but before the data is re-programmed into Flash.

An alternative method is used by the software described in this paper. This method eliminates the RAM buffer and instead requires that a minimum of two 512 byte Flash sectors are allocated to non-volatile data storage. In this method, all the non-volatile data variables are located in one or other of the Flash sectors. Whenever one non-volatile data variable is to be updated, the "new" Flash sector is erased and then all the unchanged data variables plus the new data are programmed into the "new" Flash sector. A complete set of the most recent data values always exists in Flash. The RAM requirements are greatly reduced as there is no RAM buffer. The main disadvantage of this method is that the non-volatile data variables cannot be read whilst they are being updated.

A further potential disadvantage, namely that updating a data variable causes the whole Flash block to be written, can be overcome without using excessive amounts of RAM. In order to minimise the number of Flash program/erase cycles, copies of the most frequently updated non-volatile data variables should be permanently located in RAM. The frequently changing copies in RAM can then be used to update the non-volatile data variable on a less frequent basis, prior to a power-down cycle for example.

The Flash implementation on the MC9S12C32 is programmed one word (two bytes) at a time and erased in 512 byte sectors. Refer to the Flash 32K Block Guide (S12FTS32KV1/D) for further details.

The Flash implementation on the MC9S12C32 includes an on-chip charge pump which generates the required voltages for programming and erasure, so no additional external supply is required.

A Flash block cannot be read at the same time that it is being programmed or erased. Thus the code which is controlling the program or erase operation cannot be executed from the Flash block that is being programmed or erased. The MC9S12C32 has a single Flash block, so some critical portions of the Flash programming code cannot be executed from Flash and must be executed from RAM. This code is written in assembly code in order to minimise the RAM requirement.

Program/Erase Cycles

One concern when emulating EEPROM using Flash is the issue of program/erase cycles. When EEPROM is used, each individual byte can typically be programmed and erased a finite number of times with guaranteed data integrity. Program/erase cycles of 10,000 or 100,000 are typically specified. With Flash however, the minimum erase size is a sector and the number of program/erase cycles applies to a sector. The Flash implementation on the MC9S12C32 has an erase sector size of 512 bytes and a minimum program size of one word (2 bytes). Each individual word in a sector can be programmed only once before the sector must be erased. The MC9S12C32 electrical characteristics at the time of writing guarantee a minimum of 10,000 program/erase cycles per sector. Consult the latest Data Sheet for the current electrical characteristics.

This means that, if for example the non-volatile data variables fill an entire sector (512 bytes) and 2 complete sectors are allocated to non-volatile data storage, then the total number of permitted data updates is 2 x 10,000 = 20,000. Note that this figure does not apply to each individual data variable, it is the sum of all updates for all data variables. An average number of updates per variable can be obtained by dividing the total number of permitted updates by the number of non-volatile data variables.

In a typical application, the majority of non-volatile data variables will be updated infrequently and a few will be updated more frequently. Consider an example of 256 non-volatile data variables each 2 bytes long and the minimum 2 Flash sectors allocated to non-volatile data storage. A minimum specification of 10,000 program/erase cycles per sector is assumed. If 200 of these variables are each updated 10 times in the lifetime of the microcontroller, then 200 x 10 = 2000 program/erase cycles will be used. If 50 of the remaining variables are each updated 100 times, then 50 x 100 + 2000 = 7000 program/erase cycles will be used. This would leave 20,000 - 7000 = 13,000 program/erase cycles for the remaining 6 variables, or an average of 2166 updates each.

There are two strategies that can be employed (within the context of the software described in this paper) to increase the effective permitted number of data updates. The first is simply to increase the number of Flash sectors allocated to non-volatile data storage. If the non-volatile data variables fill one entire sector, the permitted number of data updates is Number of Sectors x Program/Erase cycles per Sector.

Thus if the previous example is modified so that 16 Flash sectors are allocated to non-volatile data storage then the total number of updates becomes 16 x 10,000 = 160,000. If, as before, 200 variables are each updated 10 times and 50 variables are each updated 100 times, then the remaining 6 variables can be updated 160,000 - 2000 - 5000 = 153,000 times, or an average of 25,500 times each.

However, if the size of all the non-volatile data variables is less than half the size of a Flash sector, the Flash sector can be conceptually sub-divided into units called *banks* or pages. Each bank must be large enough to hold all the non-volatile data variables and there must be an integer number of whole banks per sector. Thus valid bank sizes are 512, 256, 128, 64, 32, 16, 8, 4, and 2 bytes. The permitted number of data updates is now:

Number of Banks per Sector x Number of Sectors x Program/Erase cycles per Sector.

If we now consider an example where there are 32 non-volatile data variables each 2 bytes long, then we can have 8 banks of 64 bytes per sector. If 4 Flash sectors are allocated to non-volatile data storage, then the permitted number of data updates for this example is 8 x 4 x 10,000 = 320,000. This equates to an average of 10,000 updates per data variable.

As mentioned previously, the number of Flash program/erase cycles used can be minimised by permanently allocating copies of the most frequently updated non-volatile data variables in RAM. The frequently changing copies in RAM can then be used to update the non-volatile data on a less frequent basis, prior to a power-down cycle for example.

Program/Erase Time

The Flash program/erase state machine is clocked from a signal derived from the microcontroller oscillator. The value of the Flash clock prescaler must be chosen so that the frequency of the Flash clock, f_{NVMOP} is within the range of 150kHz to 200kHz. The calculation of the prescaler value is performed by a pre-processor macro in the software accompanying this paper. The time required by the Flash state machine to program a single ***word*** is defined by .

$$t_{swpgm} = \frac{9}{f_{NVMOP}} + \frac{25}{f_{bus}}$$

Equation 1

Typical values for t_{swpgm} are slightly under 50µs, depending on the selected values for the oscillator frequency and the MCU internal bus frequency. Whenever a non-volatile data variable is updated, an entire bank is programmed. Thus if a bank is defined to be 32 *words*, the programming time will be approximately 50µs x 32 = 1.6ms. The execution time of the software controlling the programming process will slightly increase the actual programming time beyond this value.

The time required by the Flash state machine to erase a sector is defined by Equation 2.

$$t_{era} = \frac{4000}{f_{NVMOP}}$$

Equation 2

Typical values for t_{era} are around 21ms.

Interrupts

From the times calculated for programming and in particular erasing, it is clear that the update of non-volatile data variables could interfere with the performance of some time critical real-time applications. However the software accompanying this paper allows the possibility to service interrupts whilst the Flash is being erased or programmed. As the interrupt vector addresses normally reside in Flash and the Flash is unavailable during programming and erasure, an alternative approach is required. The solution adopted is to re-map the RAM to the top of the physical address space, so that the interrupt vectors are now located in RAM. The interrupt vector table is copied into RAM at the normal vector table address. The interrupt handler routines must also be executed from RAM and so the interrupt vectors must point to the RAM address of the interrupt handler routines. As the amount of RAM may be limited, only the most essential interrupts should remain enabled during programming and

erasure of the Flash. The Flash area which is overlaid by RAM is no longer available for use.

NOTE: If the non-maskable interrupt is enabled (X-bit cleared), then it cannot subsequently be disabled by software. In this case, the RAM must be re-mapped to the top of the memory map, the vector table must be copied to RAM, and the non-maskable interrupt service routine must be executed from RAM.

Callback Function

A less RAM intensive alternative to enabling interrupts is to enable a callback function. The callback function is called repeatedly by the ProgFlash function while the function is waiting for the program or erase operation to complete. The callback function must be executed from RAM and can access RAM and all I/O registers. The callback function is defined by the user to suit the application, typical uses for the callback function are to poll some inputs or interrupt flags or to refresh a COP watchdog.

Power Failure Recovery

One concern when programming data is the subsequent behaviour if a system failure, such as power failure, occurs during programming. In the software implementation accompanying this paper, one word of flash is reserved in each bank for status information. This status word is the last word to be programmed in each bank, and is only programmed if the whole of the rest of the bank has programmed successfully. The value of the status word indicates the most recently programmed bank. In the event of a power failure, or some other interruption to programming, a bank may be incompletely programmed and the status word will not be programmed. When power is restored and InitEeprom is next called, the last successfully programmed bank will be identified and the partially programmed bank will be ignored and subsequently erased. In this way the only data that is lost is the new data value that was to be updated at the time of the failure; this data variable retains its previous value.

Memory Map

Figure 1 illustrates the default memory map of the MC9S12C32 with a typical allocation of resources for non-volatile data storage, where interrupts are not required to be serviced during programming or erasure. Figure 2 illustrates an alternative memory map for the MC9S12C32 with the RAM remapped to the top of the memory map, to enable interrupts to be serviced during programming or erasure.

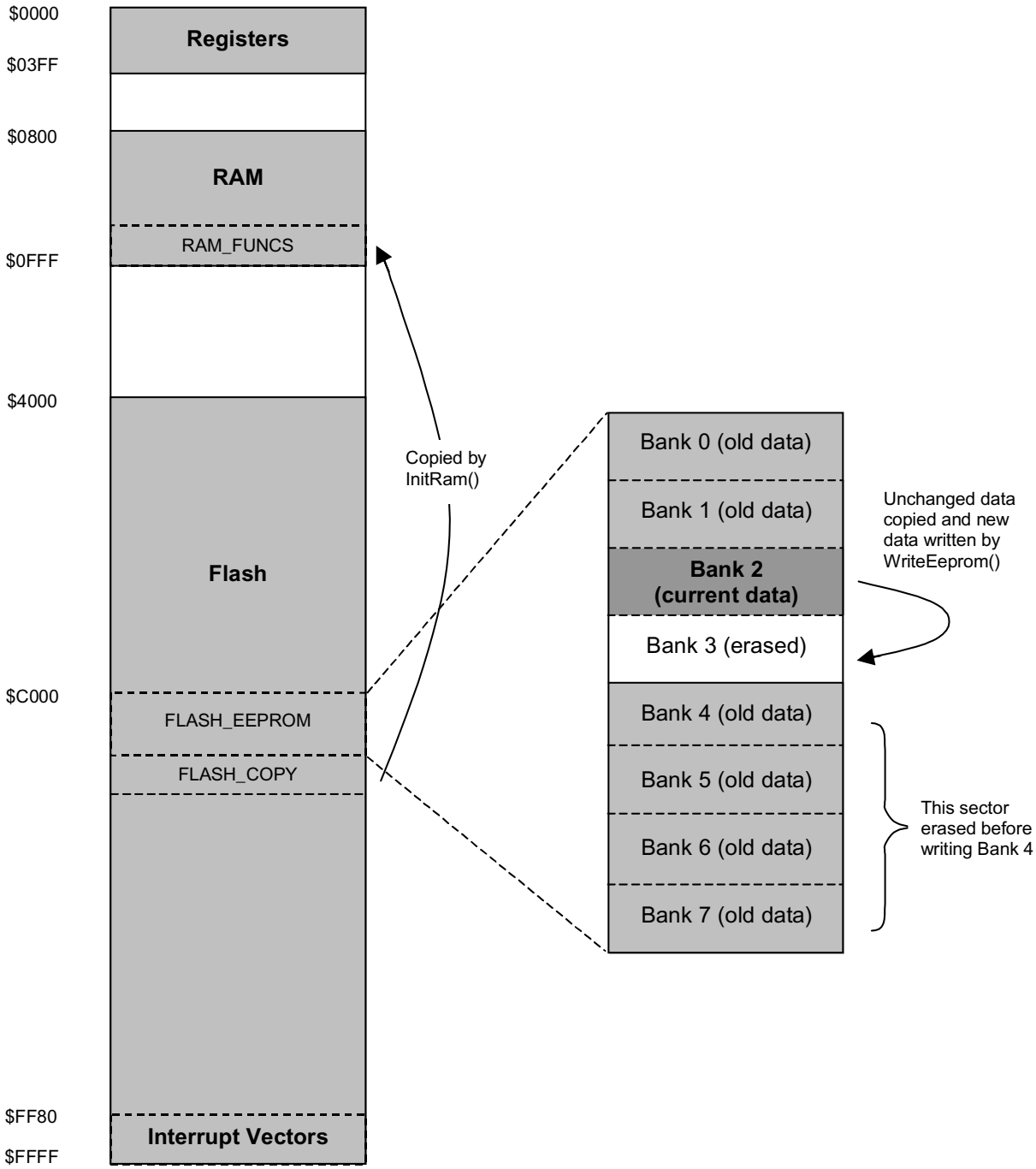

Figure 1. Example MC9S12C32 Memory Map with EEPROM Emulation

In the Figure 1 example, 2 sectors of Flash are allocated, with 8 banks of 128 bytes of non-volatile data storage. All interrupts are masked during Flash programming and erasure.

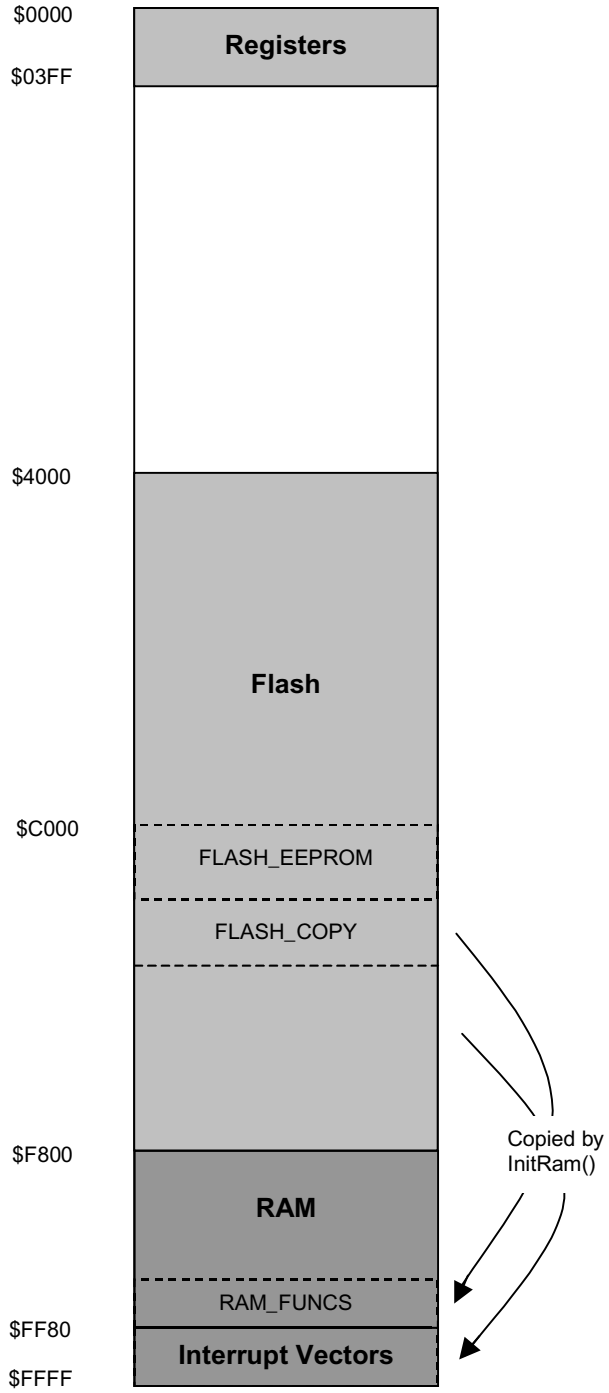

Figure 2. Example MC9S12C32 Memory Map with EEPROM Emulation and Interrupts

In the Figure 2 example, the RAM is remapped to the top of the memory map, enabling interrupts to be serviced during Flash programming and erasure.

Software Description

The purpose of each source file is described in this section. The code was written mainly in 'C' with some assembly code using the MetroWerks CodeWarrior for HCS12 tools. If a different tool suite is to be used, changes will be required to the method for identifying and linking code and data segments, the in-line assembly code would also have to be changed.

User's Application Files

Non-Volatile Data Variables

Non-volatile data variables, i.e. variables which are to be located in emulated EEPROM, are declared in the user's application files as and when required. The variables must be located in the segment called EEPROM_VARS using a #pragma statement as in the following example:

```
#pragma DATA_SEG EEPROM_VARS
unsigned char EepromVar1;
unsigned int EepromVar2;
#pragma DATA_SEG DEFAULT
```

A count should be kept of the total size of all variables allocated to EEPROM_VARS, this number will be required to determine the Eeprom bank size. The total size can also be obtained in the map file output from the linker.

In the user's application, all non-volatile data variables are accessed using the functions ReadEeprom and WriteEeprom. The user's application files which access non-volatile data variables should therefore include the file EE_Emulation.h.

EE_Emulation.h The file EE_Emulation contains function prototypes and the configuration parameters for Eeprom emulation of non-volatile data variables. This file should be included in all files which call any function in EE_Emulation.c. The following values must be correctly defined by the user:

IRQ_DURING_PROG This value can optionally be defined either in this file, or on the compiler command line. If this value is defined, it enables interrupts to be serviced during programming and erasure. The interrupt service routines for the interrupts which remain enabled during programming and erasure must be located in the file EE_RAMfuncs.c, the RAM must be re-mapped to the top of the memory map, and the vector table must be copied into the re-mapped RAM.

If IRQ_DURING_PROG is NOT defined, then it is not possible to service interrupts during programming and erasure. In this case the WriteEeprom routine takes care of masking the maskable interrupts when required.

NOTE: *If the non-maskable interrupt is enabled (X-bit cleared), then it cannot subsequently be disabled by software. If the non-maskable interrupt will be enabled (X-bit cleared), IRQ_DURING_PROG must be defined.*

EEPROM_SIZE_BYTES This value defines the number of bytes of emulated EEPROM variables. Valid values are: 2, 6, 14, 30, 62, 126, 254 and (m x 512) - 2, where m = 1, 2, 3... The selected value must be equal to or greater than the total number of bytes of all non-volatile data variables.

EEPROM_BANKS This value defines the number of banks (or copies) of non-volatile data variables that there are. EEPROM_BANKS x (EEPROM_SIZE_BYTES + 2) equals the total amount of Flash allocated to EEPROM emulation.

If EEPROM_SIZE_BYTES < 510, the total amount of Flash allocated must be 2 or more complete Flash sectors, i.e. EEPROM_BANKS x (EEPROM_SIZE_BYTES + 2) = n x 512, where permitted values of n = 2, 3, 4…

If EEPROM_SIZE_BYTES >= 510, the total amount of Flash allocated must be 2 or more banks, i.e. EEPROM_BANKS = n, where permitted values of n = 2, 3, 4…

Larger values for EEPROM_BANKS permit a larger number of updates of the non-volatile data variables, at the expense of requiring a larger amount of Flash for EEPROM emulation.

EEPROM_START This value defines the start address for the region of Flash memory that will be used to emulate EEPROM. The end of this region is given by EEPROM_START + (EEPROM_SIZE_BYTES + 2) x EEPROM_BANKS. These addresses must match the addresses of the FLASH_EEPROM section in the linker prm file.

FLASH_COPY_START This value defines the start address for the region of Flash memory where the RAM functions will be located, prior to being copied into RAM. The end address is FLASH_COPY_START + RAM_FUNCS_SIZE - 1. These addresses must match the addresses of the FLASH_COPY section in the linker prm file.

RAM_FUNCS_START This value defines the start address for the region of RAM where the RAM functions will be copied to. The end address is RAM_FUNCS_START + RAM_FUNCS_SIZE - 1. These addresses must match the addresses of the RAM_FUNCS section in the linker prm file.

RAM_FUNCS_SIZE This value defines the number of bytes required by the RAM functions. This value may be obtained from the map file obtained after compiling and linking the EE_RAMfuncs.c file. A minimum of 34 bytes is required for the case where IRQ_DURING_PROG is not defined and EECALLBACK is not defined. In the case where either IRQ_DURING_PROG or EECALLBACK are defined, the value of RAM_FUNCS_SIZE will depend on the size of the user defined functions and can be obtained from the map file obtained when compiling and linking the file EE_RAMFuncs.c.

VT_START This value is only required if interrupts must be serviced during programming (IRQ_DURING_PROG defined). This value defines the start address for the region of (re-mapped) RAM that the vector table will be copied to. This value will normally be 0xFF80.

VT_SIZE This value is only required if interrupts must be serviced during programming (IRQ_DURING_PROG defined). This value matches the size of the VectorTable array and will normally be 126.

EE_Emulation.c The file EE_Emulation contains the functions required for Eeprom emulation of non-volatile data variables which are executed from Flash. All functions in this file are located in the DEFAULT code segment. The user should not modify this file.

ReadEeprom The ReadEeprom function is used to obtain the current value of a non-volatile data variable. This is the only way to obtain the current value, accessing the data variable directly will NOT generally result in the current value being read. InitEeprom must have been called once before this function is called.

Prototype: void ReadEeprom(void *srcAddr, void *destAddr, UINT16 size)

Parameters:

srcAddr	pointer to the non-volatile data variable to be read.
destAddr	pointer to a RAM location to copy the read value to.
size	size in bytes of the non-volatile data variable to be read.

Return: void

Example:

```
ReadEeprom(&EepromVar1, ReadBuffer, sizeof(EepromVar1));
```

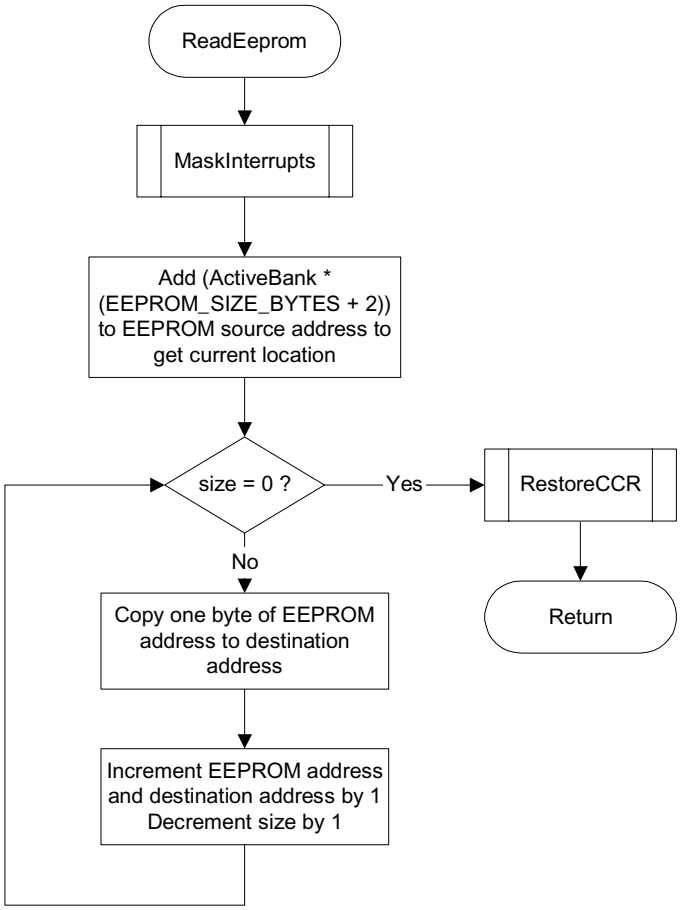

Figure 3. ReadEeprom Flow Diagram

WriteEeprom

The WriteEeprom function is used to update a non-volatile data variable with a new value. InitEeprom must have been called once before this function is called. Whenever this function is called, a whole new bank is programmed. This will take approximately (EEPROM_SIZE_BYTES + 2) x 25µs. Furthermore, if there are no more erased banks in the current sector, then the next Flash sector will be erased, this will take approximately 20ms. If the macro IRQ_DURING_PROG is not defined, this routine will take care of masking interrupts when required. InitEeprom must have been called once before this function is called.

Prototype: UINT8 WriteEeprom(void *destAddr, void *srcAddr, UINT16 size)

Parameters:

destAddr	pointer to the non-volatile data variable to be updated.
srcAddr	pointer to the new data to be written to the non-volatile data variable.
size	size in bytes of the non-volatile data variable to be updated.

Return:

PASS	update was successful.
FAIL	a programming failure occurred during the update process.

Example: status = WriteEeprom(&EepromVar1, WriteBuffer, sizeof(EepromVar1));

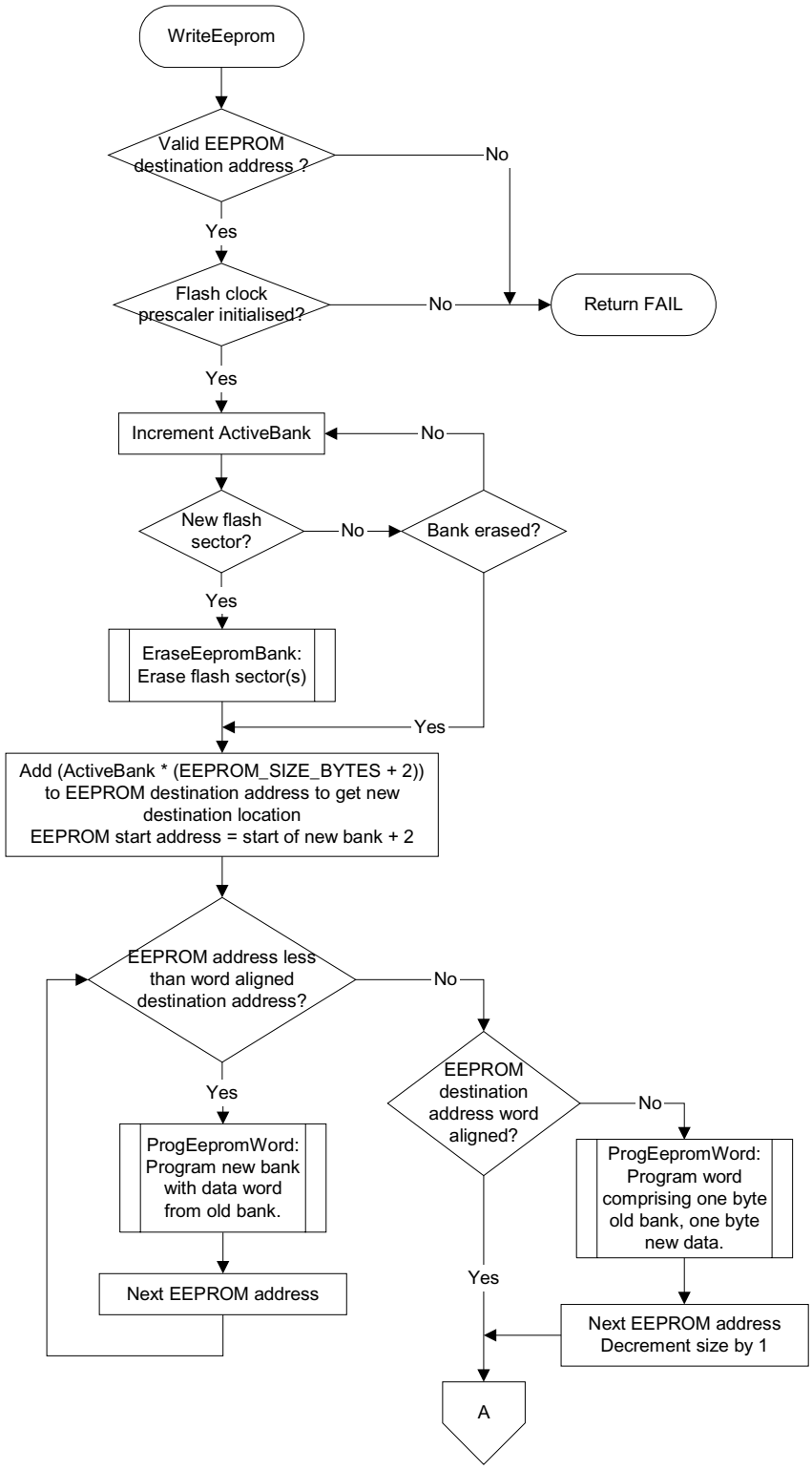

Figure 4. WriteEeprom Flow Diagram

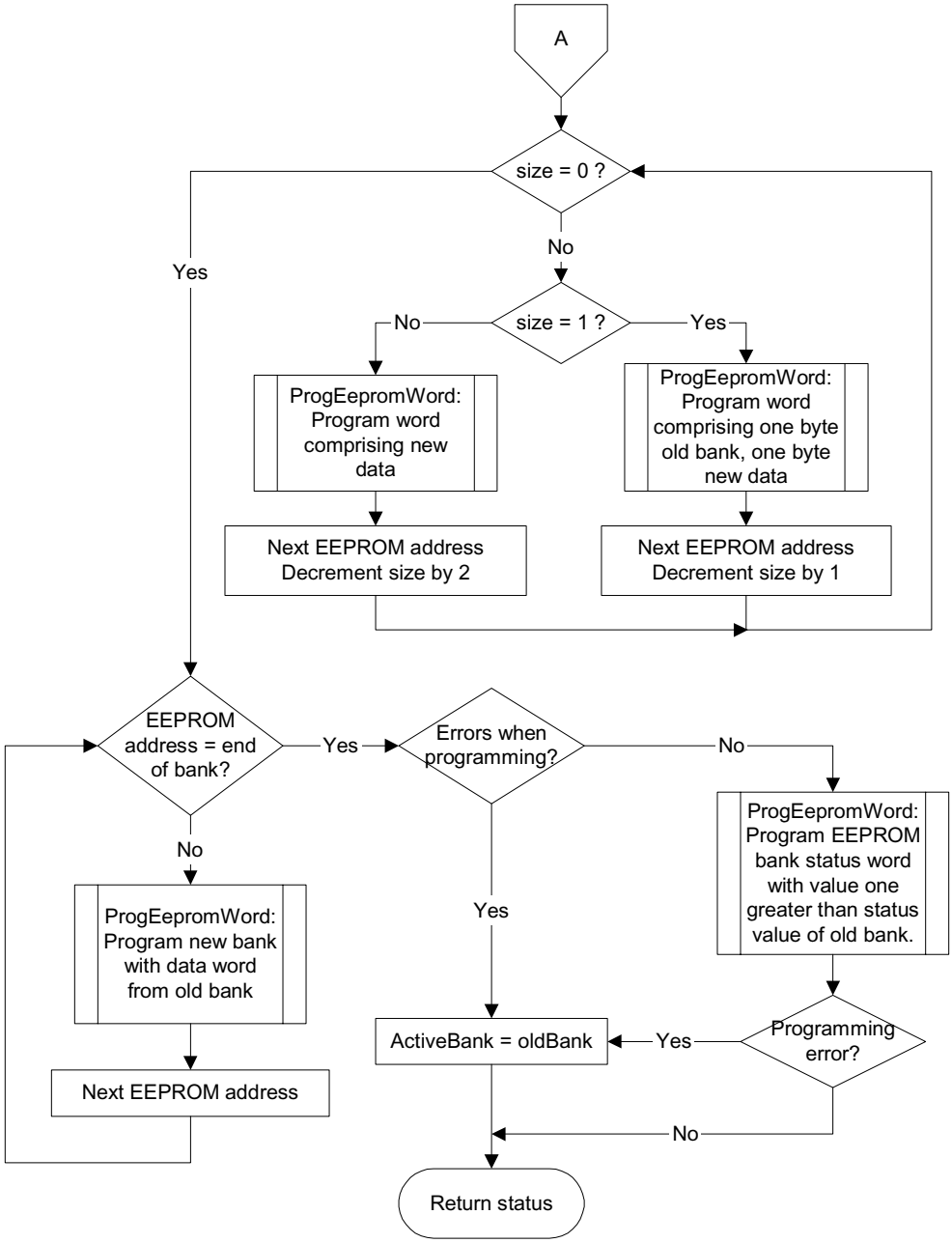

Figure 5. WriteEeprom Flow Diagram (continuation)

InitEeprom This function must be called once before either ReadEeprom or WriteEeprom are called. This function initialises the Flash prescaler, copies the required functions to RAM and determines the bank with the most recent data.

Prototype: void InitEeprom(void)

Parameters: void

Return: void

Example: InitEeprom();

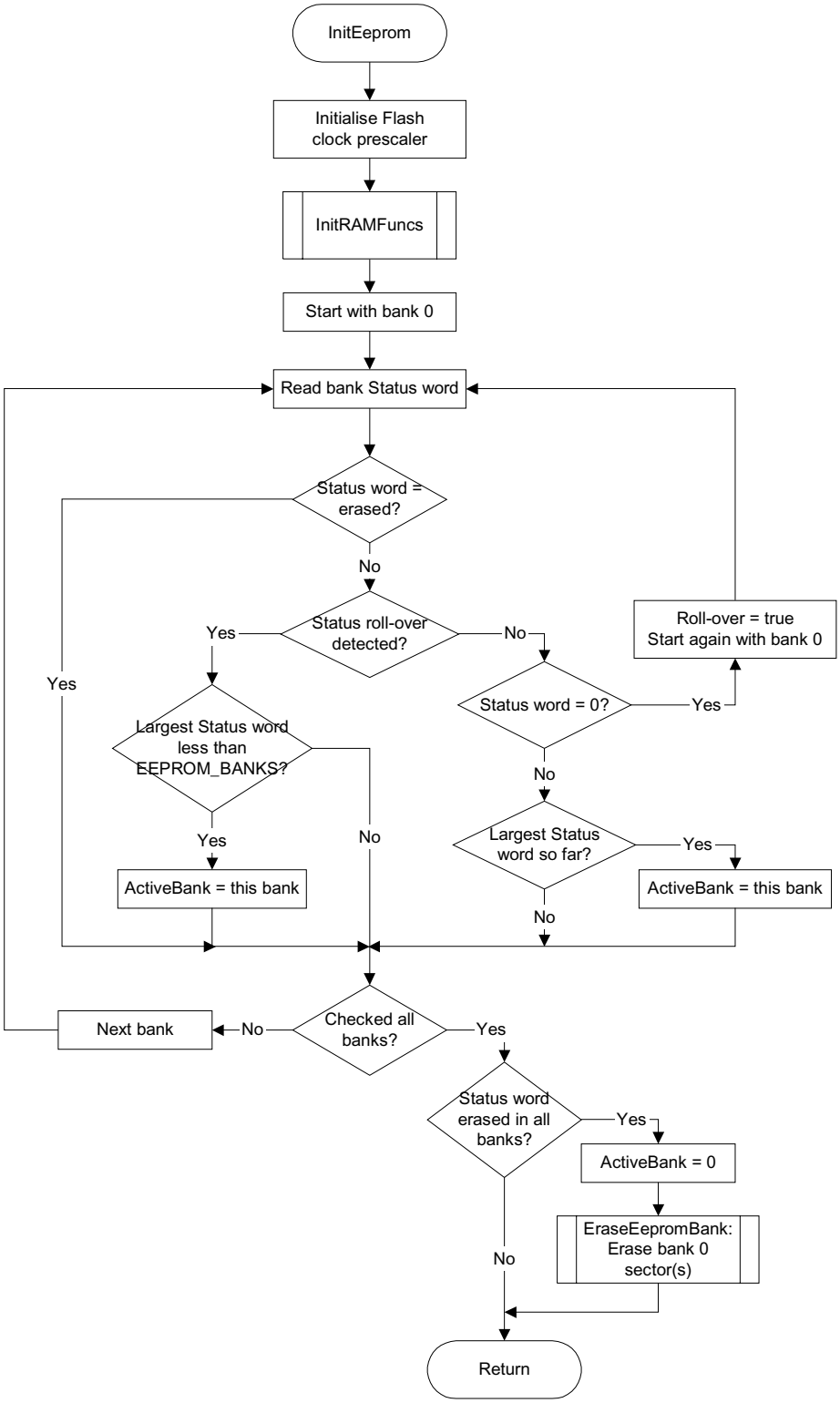

Figure 6. InitEeprom Flow Diagram

InitRAMFuncs This function is called by InitEeprom to copy the required functions to RAM. If the macro IRQ_DURING_PROG is defined, the vector table is also copied into addresses 0xFF80 to 0xFFFD, which is expected to be RAM. This function is provided to enable the application to refresh the code in RAM if this is considered necessary.

Prototype: void InitRAMFuncs(void)

Parameters: void

Return: void

Example: InitRAMFuncs();

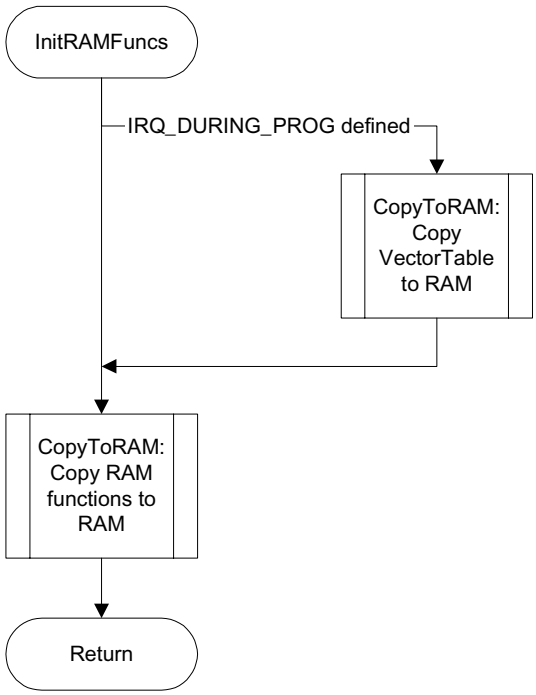

Figure 7. InitRAMFuncs Flow Diagram

CopyToRAM This static function is called by InitRAMFuncs to copy code and data from Flash to RAM. It is not callable by the application.

Prototype: void CopyToRAM(UINT8 *src, UINT8 *dest, UINT16 size)

Parameters:

src	pointer to the source code to be copied.
dest	pointer to the destination for the code to be copied to.
size	size in bytes of code to be copied.

Return: void

Example:
```
CopyToRAM((UINT8 *)FLASH_COPY_START,
(UINT8 *)RAM_FUNCS_START, RAM_FUNCS_SIZE);
```

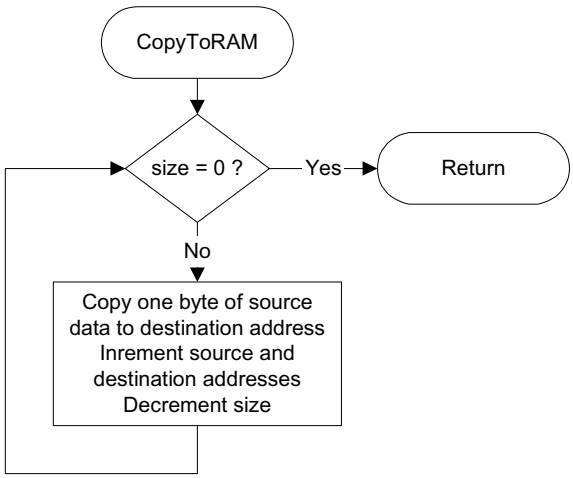

Figure 8. CopyToRAM Flow Diagram

ProgEepromWord This static function is called by WriteEeprom to program a single word of emulated EEPROM. After verifying the programmed word, the flash address pointer is incremented. Interrupts are masked if IRQ_DURING_PROG is not defined. This function is not callable by the application.

Prototype UINT8 ProgEepromWord(UINT16 **progAddr, UINT16 data);

Parameters:

progAddr	pointer to a pointer to the flash location to be programmed.
data	data word to be programmed

Return

PASS	word was programmed successfully
FAIL	word failed to program

Example:
```
Status = ProgEepromWord(&eepromAddr, eepromData);
```

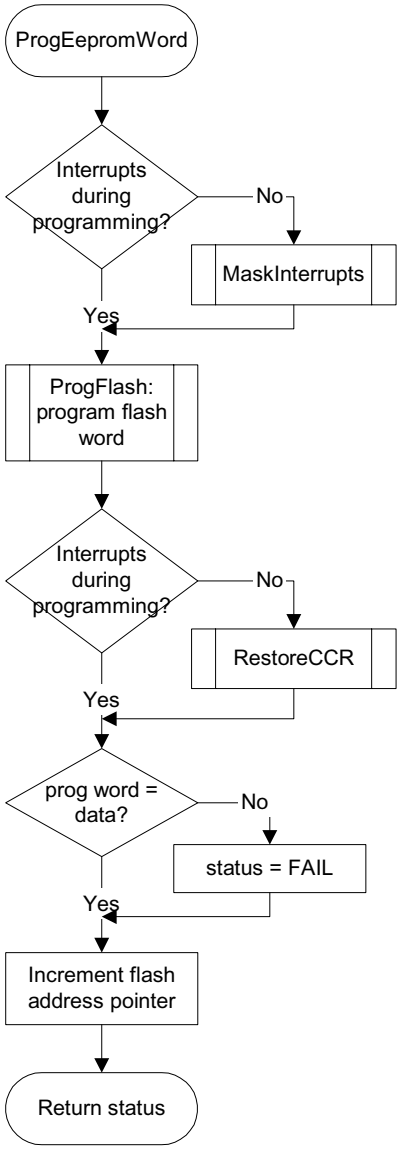

Figure 9. ProgramEepromWord Flow Diagram

EraseEepromBank This static function is called to erase a bank of emulated EEPROM. If the EEPROM bank size is less than or equal to the size of a flash sector, one sector is erased. If the EEPROM bank size is greater than the flash sector size, then as many sectors as required are erased. Each sector takes approximately 20ms. Interrupts are masked if IRQ_DURING_PROG is not defined. This function is not callable by the application.

Prototype: UINT8 EraseEepromBank(UINT16 *eepromAddr);

Parameters:	eepromAddr	address of the (first) flash sector(s) to be erased
Return	PASS	flash erase commands executed successfully
	FAIL	flash erase commands failed

Example: status = EraseEepromBank(eepromAddr);

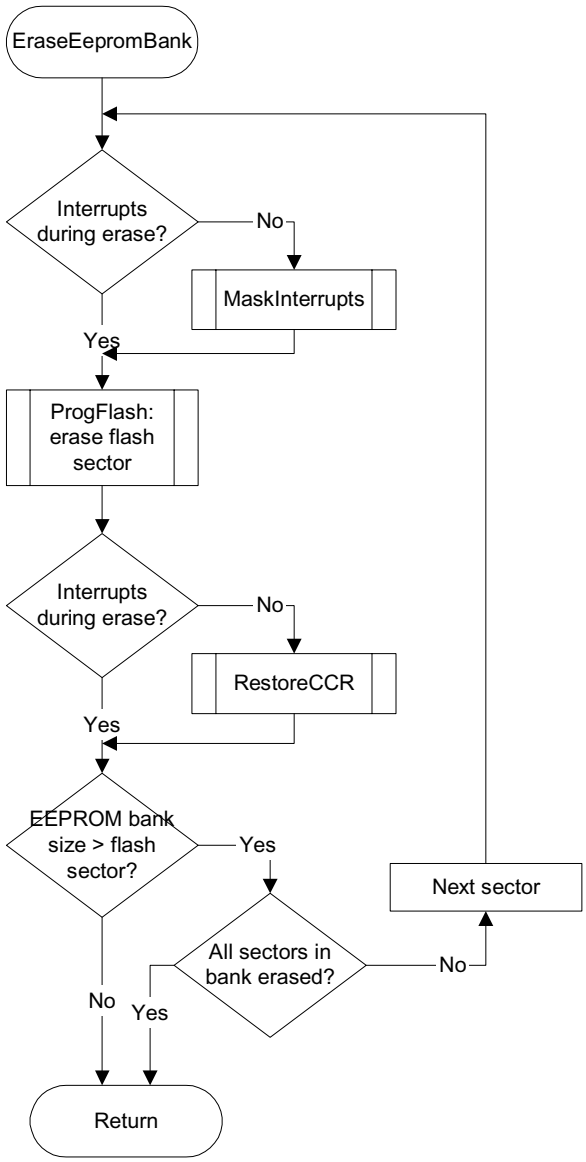

Figure 10. EraseEepromBank Flow Diagram

MaskInterrupts This static function is called to mask interrupts before programming and erasing, if IRQ_DURING_PROG is not defined.

Prototype: void MaskInterrupts(UINT8 *dest);

Parameters: dest pointer to storage location for CCR

Return: void

Example: MaskInterrupts(&CCRCopy);

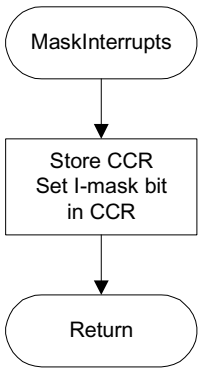

Figure 11. MaskInterrupts flow diagram

RestoreCCR This static function is called to restore the CCR to its previous value after programming and erase, if IRQ_DURING_PROG is not defined.

Prototype: void RestoreCCR(UINT8 src)

Parameters: src value to store in CCR

Return void

Example: RestoreCCR(CCRCopy);

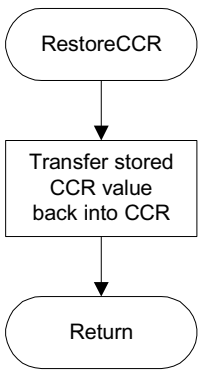

Figure 12. RestoreCCR flow diagram

EE_RAMFuncs.h

The file EE_Emulation contains function prototypes for the files in EE_RAMFuncs.c. Remember to add function prototypes for any user defined interrupt service routines added to EE_RAMFuncs.c. EECALLBACK should be defined in this file to enable EECallBackFunc. This file is included in EE_Emulation.c and VectorTable.c.

EE_RAMfuncs.c

This file should contain all functions which are required to execute from RAM while the Flash is being programmed or erased. These will include:

ProgFlash - always required,

EECallBackFunc - required if EECALLBACK defined.

In addition, if IRQ_DURING_PROG is defined, all user defined interrupt service routines for interrupts which will remain enabled during programming and erasure must be located in this file. Interrupt service routines for non-essential interrupts which are disabled during programming and erasure should be located in Flash and not in this file i.e. not in the code segment RAM_FUNCTIONS.

All functions in this file are located in the code segment RAM_FUNCTIONS. This file is compiled and linked to its final RAM address as a ROM library prior to being linked into the final application with EE_Emulation.c. This ensures that all calls to functions within this file result in calls to the final execution addresses, which is more efficient than calling the functions using function pointers. The ROM library is then appended to the final application file with an offset resulting in a Flash address. This process is described in the Example Project section.

ProgFlash This function is copied to and executed from RAM. This function is written in assembly code for maximum code efficiency, to minimise the RAM requirement. This function performs programming of a single word or erasure of a single sector of Flash. Programming a word requires approximately 50µs. Erasing a sector requires approximately 20ms. This function is called by ProgEepromWord and EraseEepromBank and is not callable by the application. If EECALLBACK is defined, this function will repeatedly call the user defined function EECallBackFunc while programming or erasure is taking place.

Prototype: UINT8 ProgFlash(UINT8 command, UINT16 *progAddr, UINT16 data)

Parameters:

command	PROG - programs a single word of data,
	ERASE - erases a single Flash sector.
progAddr	pointer to the Flash word to be programmed or the Flash sector to be erased.
data	the data word to be programmed, dummy data if erasing.

Return:

PASS	the ACCERR or PVIOL bits were not set when the program or erase command was executed.
FAIL	the ACCERR or PVIOL bits were set when the program or erase command was executed.

Example : status = ProgFlash(PROG, eepromAddr, eepromData.word);

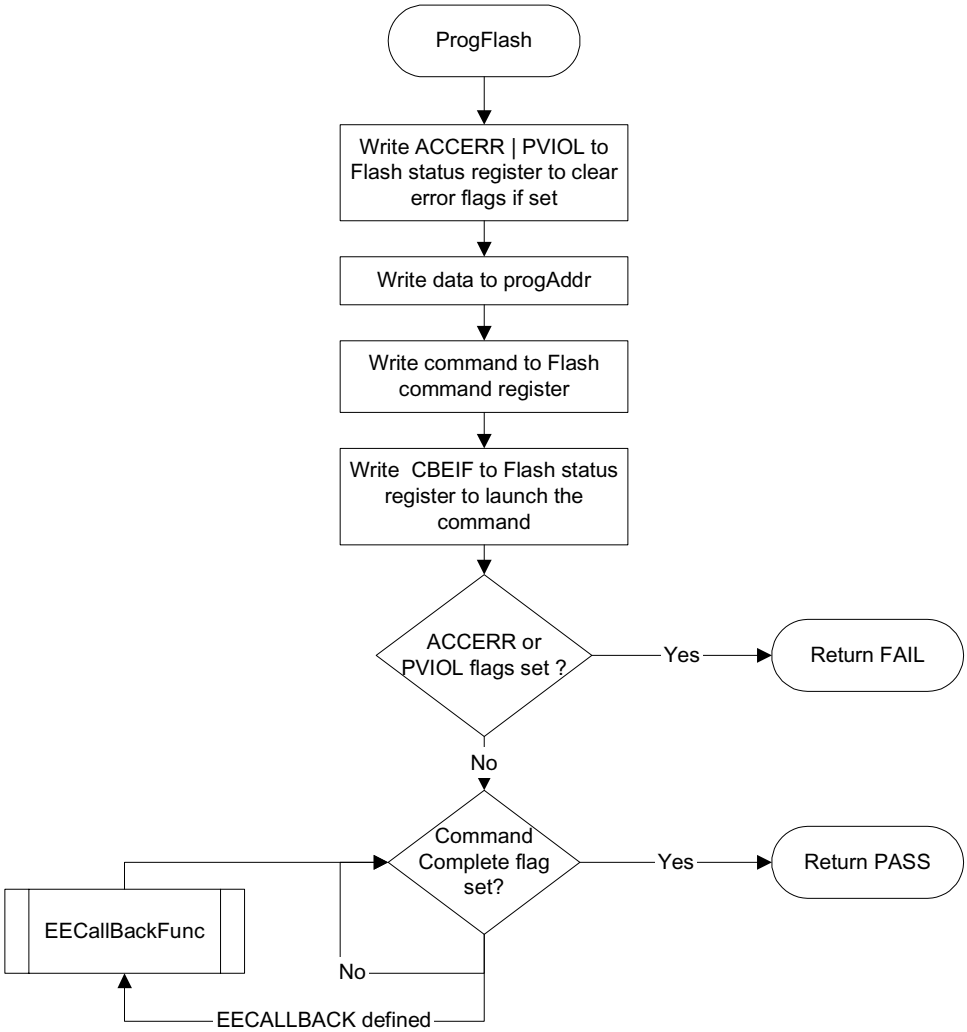

Figure 13. ProgFlash Flow Diagram

EECallBackFunc This is a user defined function which is enabled by defining EECALLBACK in EE_RAMfuncs.h. If enabled, this function is called repeatedly by ProgFlash while waiting for the CCIF flag to be set during program and erase operations. This function is copied to and executed from RAM. This function could be used to refresh a watchdog for example.

Prototype: void EECallBackFunc(void)

Parameters: void.

Return: void.

Example : EECallbackFunc(void);

VectorTable.c This file is only required if interrupts must be serviced during programming (IRQ_DURING_PROG defined). This file contains the vector table that is copied into re-mapped RAM. This vector table must contain the addresses of ALL interrupt service routines - those which will be located in RAM and those located in Flash which are disabled during programming. The reset vector is not required in this table and must be located at address 0xFFFE in Flash, as the re-mapped RAM will be moved back to its default location during a reset.

Start12.c This is the default start up file included with the Metrowerks Codewarrior stationery that has been modified slightly. This modification is only required if interrupts must be serviced during programming (IRQ_DURING_PROG defined). In this case, the INITRM register is initialised with 0xF9 to re-map the RAM to the top of the memory map.

mcucfg.h This file contains some general MCU configuration data that is required for correct determination of macros etc.

OSCCLK_FREQ_KHZ This value defines the frequency of the MCU oscillator in kHz. The defined value must have an "L" appended (e.g. 8000L) to force a long type so that the FCLK_PRESCALER macro works correctly.

PLL_ENABLED This value should be defined if the PLL will be enabled. PLL initialisation code is not included in this example.

REFDV If PLL_ENABLED is defined, the value for the REFDV register should be defined so that the BUSCLK_FREQ_KHZ macro can be evaluated.

SYNR	If PLL_ENABLED is defined, the value for the SYNR register should be defined so that the BUSCLK_FREQ_KHZ macro can be evaluated.
TIMER_PRESCALER_FACTOR	This value is only required for the example if interrupts must be serviced during programming (IRQ_DURING_PROG defined). This value defines the Timer Prescaler Factor, permitted values are 1, 2, 4, 8, 16, 32, 64, and 128.
mottypes.h	This file contains type definitions for common types such as unsigned char, signed char, etc.
s12_fectl.h	This file contains definitions for the Flash register structure as well as the FCLK_PRESCALER macro. This macro is essential for the correct initialisation of the Flash clock prescaler and depends on correct definitions of OSCCLK_FREQ_KHZ and BUSCLK_FREQ_KHZ in mcucfg.h.
s12_stdtimer.h	This file contains definitions for the standard timer register structure used on the MC9S12C32. This file is only required for the example if interrupts must be serviced during programming (IRQ_DURING_PROG defined).

Example Project

A Metrowerks Codewarrior example project is available for download to accompany this paper. The pertinent features of this project are explained here, along with instructions on how to build the project.

The project is split into 4 targets, 2 for the case where interrupts are masked during programming and erasure, and 2 for the case where interrupts remain enabled during programming and erasure.

In both cases, building the final application is a 2 step process. First the RAM functions are built as a ROM library. The second step is to build the final application linked to the ROM library and with the ROM library appended to the final absolute or s-record file.

Example with Interrupts Masked during Programming	Two targets are provided to build an example with interrupts masked during programming and erasure.

EE RAM functions, no IRQ

This target is provided to build the RAM functions as a ROM library. The file EE_RAMfuncs.c is compiled and linked by itself, generating the absolute file EE_RAMfuncs.abs and the s-record file EE_RAMfuncs.sx.

The linker directive "-AsROMlib" is specified on the linker command line to enable this to be built as a ROM library, so no main function is expected. The usual start-up initialisation structure is not generated either.

The linker directive "-M" is specified on the linker command line so that a map file is generated.

The linker directive "-B" is specified on the linker command line so that an s-record file is generated.

The ROM library is placed at the addresses specified in the associated prm file, EERAMfuncs.prm, listed below:

```
NAMES
END

SECTIONS
        LASH_REGS           = NO_INIT             0x100  TO 0x110;
        DUMMY_RAM           = READ_WRITE          0x800  TO 0x800;
        RAM_FUNCS           = READ_ONLY           0x0FD0 TO 0xFFF;
END

PLACEMENT
        DEFAULT_RAM                               INTO  DUMMY_RAM;
        DEFAULT_ROM,RAM_FUNCTIONS                 INTO  RAM_FUNCS;
        FLASH_REG                                 INTO  FLASH_REGS;
END

ENTRIES
ProgFlash                                         /* list all RAM function names here */
                                                  /* eg EECallBackFunc */
END
```

The RAM functions defined within code segment RAM_FUNCTIONS are placed in section RAM_FUNCS. This is located at the RAM address at which the functions will be copied to and executed from, this section must have the attribute READ_ONLY. The addresses allocated for the RAM functions may need to be expanded if EECALLBACK is defined.

The structure containing the Flash control registers is placed in the section FLASH_REGS which is located at the default address. If the registers are remapped after reset then this address will have to be changed. This section must have the attribute NO_INIT.

The pre-defined segments DEFAULT_RAM and DEFAULT_ROM are listed to comply with the linker, although they contain no data or code.

Finally, all the RAM function names that are not explicitly called must be listed under the ENTRIES command.

EE Emulation no IRQ

This target is provided to build the final application with interrupts disabled during programming and erasure.

The files EE_Emulation.c and main.c are compiled and linked together with the ROM library generated by the "EE RAM functions, no IRQ" target, generating the absolute file EE_Emulation.abs.

The linker directive "-AddEE_RAMfuncs.abs" is specified on the linker command line to link in the previously generated ROM library.

The linker directive "-M" is specified on the linker command line so that a map file is generated.

The application is linked to the addresses specified in the associated prm file, EE_Emulation.prm, listed below:

```
NAMES
END

SECTIONS
         FLASH_REGS          = NO_INIT             0x100  TO 0x110;
         RAM                 = READ_WRITE          0x0800 TO 0x0FCF;
                                                   /* reserve for RAM_FUNCS */
         RAM_FUNCS           = NO_INIT             0x0FD0 TO 0x0FFF;
                                                   /* min 2 complete sectors */
         FLASH_EEPROM        = NO_INIT             0xC000 TO 0xC3FF;
         FLASH_CODE          = READ_ONLY           0xD030 TO 0xD3FF;
END

PLACEMENT
         PRESTART, STARTUP,
         ROM_VAR, STRINGS,
         NON_BANKED, DEFAULT_ROM,
         COPY                                      INTO  FLASH_CODE;
EEPROM_STAT, EEPROM_VARS                           INTO  FLASH_EEPROM; /* EEPROM_STAT must be first */
         DEFAULT_RAM                               INTO  RAM;
         FLASH_REG                                 INTO  FLASH_REGS;
END

STACKSIZE 0x100

VECTOR ADDRESS 0xFFFE  _Startup                    /* list all other vectors here */

                                                   /* 0xC030 + 0x0FD0 = 0xD000 = FLASH_COPY_START */
HEXFILE EE_RAMfuncs.sx OFFSET 0xC030
```

The application code is placed in section FLASH_CODE.

The section FLASH_EEPROM is reserved for EEPROM emulation data and contains all the variables in segment EEPROM_VARS. This section must be large enough for EEPROM_BANKS x (EEPROM_SIZE_BYTES + 2).

The structure containing the Flash control registers is placed in the section FLASH_REGS that is located at the default address. If the registers are

remapped after reset then this address will have to be changed. This section must have the attribute NO_INIT.

The section RAM_FUNCS is reserved as a space in RAM for the RAM functions to be copied to for execution. The RAM functions are copied here from the FLASH_COPY section by the InitRAMFuncs function. The addresses allocated for the RAM functions may need to be expanded if EECALLBACK is defined.

Finally, the EE_RAMFuncs ROM library file must be included using the linker HEXFILE command. An OFFSET must be added to the addresses in this file so that it is loaded into Flash. The offset is chosen so that when added to the original addresses the desired Flash address is obtained. This address is equivalent to FLASH_COPY_START in EE_Emulation.h. No other code must be located in this address space.

Example with Interrupts Enabled during Programming

Two targets are provided to build an example with interrupts enabled during programming and erasure.

EE RAM functions with IRQs

This target is provided to build the RAM functions as a ROM library. The file EE_RAMfuncs.c is compiled and linked by itself, generating the absolute file EE_RAMfuncsIRQ.abs and the s-record file EE_RAMfuncsIRQ.sx.

The compiler directive "-DIRQ_DURING_PROG" is specified on the compiler command line to activate the code required to enable interrupts to be serviced.

The linker directive "-AsROMlib" is specified on the linker command line to enable this to be built as a ROM library, so no main function is expected. The usual start-up initialisation structure is not generated either.

The linker directive "-M" is specified on the linker command line so that a map file is generated.

The linker directive "-B" is specified on the linker command line so that an s-record file is generated.

The ROM library is placed at the addresses specified in the associated prm file, EERAMfuncsIRQ.prm, listed below:

```
NAMES
END

SECTIONS
         TIMER_REGS          = NO_INIT             0x0040 TO 0x006F;
         FLASH_REGS          = NO_INIT             0x100  TO 0x110;
         DUMMY_RAM           = READ_WRITE          0xF800 TO 0xF800;
         RAM_FUNCS           = READ_ONLY           0xFF00 TO 0xFF7F;
END

PLACEMENT
         DEFAULT_RAM                               INTO  DUMMY_RAM;
         DEFAULT_ROM,RAM_FUNCTIONS                 INTO  RAM_FUNCS;
         FLASH_REG                                 INTO  FLASH_REGS;
         TIMER_REG                                 INTO  TIMER_REGS;
END

ENTRIES
ProgFlash                                          /* list all RAM function names here */
TIMER0_ISR                                         /* list all RAM ISR functions names here */
END
```

The RAM functions defined within code segment RAM_FUNCTIONS are placed in section RAM_FUNCS. This is located at the RAM address at which the functions will be copied to and executed from, this section must have the attribute READ_ONLY. The addresses allocated for the RAM functions may need to be expanded, depending on the size of the interrupt service routines and if EECALLBACK is defined.

The structure containing the Flash control registers is placed in the section FLASH_REGS which is located at the default address. If the registers are remapped after reset then this address will have to be changed. This section must have the attribute NO_INIT.

This example uses the timer module, so the structure containing the timer control registers is placed in the section TIMER_REGS at the default address.

The pre-defined segments DEFAULT_RAM and DEFAULT_ROM are listed to comply with the linker, although they contain no data or code.

Finally, all the RAM function names and interrupt service routines that are executed from RAM and are not explicitly called must be listed under the ENTRIES command.

EE Emulation with IRQs

This target is provided to build the final application with interrupts enabled during programming and erasure.

The files EE_Emulation.c, VectorTable.c and main.c are compiled and linked together with the ROM library generated in the "EE RAM functions with IRQs" target, generating the absolute file EE_EmulationIRQ.abs.

The compiler directive "-DIRQ_DURING_PROG" is specified on the compiler command line to activate the code required to enable interrupts to be serviced.

The linker directive "-AddEE_RAMfuncsIRQ.abs" is specified on the linker command line to link in the previously generated ROM library.

The linker directive "-M" is specified on the linker command line so that a map file is generated.

The application is linked to the addresses specified in the associated prm file, EE_Emulation_IRQ.prm, listed below:

```
NAMES
END

SECTIONS
         TIMER_REGS          = NO_INIT             0x0040 TO 0x006F;
         FLASH_REGS          = NO_INIT             0x0100 TO 0x010F;
         RAM                 = READ_WRITE          0xF800 TO 0xFEFF;
                                                   /* reserve for RAM_FUNCS */
         RAM_FUNCS           = NO_INIT             0xFF00 TO 0xFF7F;
                                                   /* min 2 complete sectors */
         FLASH_EEPROM        = NO_INIT             0xC000 TO 0xC3FF;
         FLASH_CODE          = READ_ONLY           0xD100 TO 0xD7FF;
END

PLACEMENT
         PRESTART, STARTUP,
         ROM_VAR, STRINGS,
         NON_BANKED, DEFAULT_ROM,
         COPY                          INTO  FLASH_CODE;
         RAM_FUNCTIONS                 INTO  FLASH_COPY;
         EEPROM_STAT, EEPROM_VARS      INTO  FLASH_EEPROM;  /* EEPROM_STAT must be first */
         DEFAULT_RAM                   INTO  RAM;
         FLASH_REG                     INTO  FLASH_REGS;
         IMER_REG                      INTO  TIMER_REGS;

END

STACKSIZE 0x100

ENTRIES
VectorTable
END

VECTOR ADDRESS 0xFFFE  _Startup                  /* reset vector only */

                                                 /* 0xFFFFD100 + 0xFF00 = 0xD000 = FLASH_COPY_START */
HEXFILE EE_RAMfuncsIRQ.sx OFFSET 0xFFFFD100
```

The application code is placed in section FLASH_CODE.

The section FLASH_EEPROM is reserved for EEPROM emulation data and contains all the variables in segment EEPROM_VARS. This section must be large enough for EEPROM_BANKS x EEPROM_SIZE_BYTES.

The structure containing the Flash control registers is placed in the section FLASH_REGS that is located at the default address. If the registers are remapped after reset then this address will have to be changed. This section must have the attribute NO_INIT.

This example uses the timer module, so the structure containing the timer control registers is placed in the section TIMER_REGS at the default address.

The section FLASH_COPY is reserved as a space for the RAM functions in Flash. The RAM functions are placed at this address using the linker HEXFILE command to load in the previously generated ROM library EE_RAMfuncsIRQ.sx, with an OFFSET added to each record. The OFFSET value of 0xFFFF4100 is calculated by subtracting the RAM_FUNCS start address from the FLASH_COPY start address.

The section RAM_FUNCS is reserved as a space in RAM for the RAM functions to be copied to for execution. The addresses allocated for the RAM functions may need to be expanded, depending on the size of the interrupt service routines and if EECALLBACK is defined. The RAM functions are copied here from the FLASH_COPY section by the InitRAMFuncs function.

Finally, the EE_RAMFuncsIRQ ROM library file must be included using the linker HEXFILE command. An OFFSET must be added to the addresses in this file so that it is loaded into Flash. The offset is chosen so that when added to the original addresses the desired Flash address is obtained. This address is equivalent to FLASH_COPY_START in EE_Emulation.h. No other code must be located in this address space.

Debugging Tips

When debugging the examples, the users final application, or any other code that writes to Flash, it is essential that the debugger does not attempt to effect a breakpoint by attempting to write a SWI instruction to a Flash address. The write to Flash by the debugger is interpreted as the beginning of a Flash programming sequence by the Flash state machine, leading to a Flash access error when the Flash programming routine is executed.

When using the Metrowerks HI-WAVE debugger, SWI breakpoints are prevented by setting the debugger option:

HWBREAKONLY ON

This may be done on the debugger command line, or within the startup.cmd file.

When set to ON, this option prevents the debugger from attempting to write the SWI instruction to effect a breakpoint. Instead, the hardware breakpoint module is used exclusively.

A current limitation of the Metrowerks HI-WAVE debugger is that it is only able to load symbols from a single ELF executable file at a time.

In order to debug the code in the ROM library, the following steps must be followed:

- Delete all commands from the preload.cmd and postload.cmd files.
- Put a breakpoint on the call to the ROM library function,
- Make a single step to enter into the ROM library code. The assembly window should display some code, but the Source and Data 1 window will be empty.
- In the Target menu select Load.
- In the "Load Executable File" dialog, browse for the ROM library executable file and make sure the "Load Symbol Only" radio button is checked.
- Click on "Open". The symbolic information from the ROM library should be loaded into the debugger and you should be able to debug your ROM library.

When returning to the main application, repeat the above process to re-load the symbol table for the main application again.

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

Motorola Japan Ltd.; SPS, Technical Information Center,
3-20-1, Minami-Azabu Minato-ku, Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.;
Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors