Eeprom Emulation Driver

Subject: Module specification

Revision: 1.0
Author:
Last Updated: 31/01/2005 02:33
Last Updated By:
Printout Date: 14/02/2006 02:56
File: H:\NV\EE_drv_ModuleSpecification.doc

History

Date	Author	Revision	Comments
2004-01-31	PS	1.0	EEPROM Specification rev. 0.8 split into module spec. and interface spec. Various updates.

Terminology

Dynamic Parameters	Parameters that might be changed during runtime in normal operation mode
Static Parameters	Parameters that are adjusted in production and which are different from MS to MS. These parameters are never changed after the MS has left production.
Default Parameters	Parameters that are set in production which are the same for all MS's, and which will never be changed after the MS has left production.

Contents

1 Introduction

In a Mobile Station (MS) there is a need for parameters, which can be stored in none volatile memory (Eeprom), so the data contents is not lost when the MS is switched off. The parameters can be split up in three groups:

- Parameters that might be changed during runtime in normal operation mode – These are defined as *Dynamic parameters.*
- Parameters that are adjusted in production and which are different from MS to MS. These parameters are never changed after the MS has left production. These are defined as *Static parameters.*
- Parameters that are set in production which are the same for all MS's, and which will never be changed after the MS has left production. These are defined as *Default static parameters.*

Only the *Dynamic parameters* and the *Static parameters* will be described in this document, the *Default static parameters* will be treated as program code and will be programmed together with the SW package during software download.

1.1 Overview

In this document the emulation of the Eeprom in flash is described. The main topics are: how to integrate access to the flash which renders the code in the flash un-executable during access, handling power losses, how data is stored in the flash, and which features are emulated.

1.2 Module contents

File	Description
ee.c	This file contains the eeprom emulation code.
ee.h	This file defines the interface function to the emulation and the location of the blocks in the memory map.
eep.c	This file defines the default data in the structures used in the eeprom emulation. (file is partly generated by EEP tool)
eep.h	This file defines the data type structures used in the eeprom emulation. (file is partly generated by EEP tool)

2 Architecture

In this chapter the detailed design of the Eeprom Emulation Driver is described.

2.1 Definition of concept

To save components and cost the Eeprom is emulated in flash instead of using a dedicated Eeprom-device. The emulation courses some problems, because it is not possible after read to write (and vice versa) to the flash without executing some code that physically change the direction first[1]. This means, that all interrupt entries associated to the interrupt that occurs during write to flash has to be located in RAM, at least the part of the code that is executed before the suspend/resume part. Time critical interrupt that can't allow overhead processing time due to suspend/resume might need to be located fully in RAM. The complexity of the emulation-SW is high, because the impact on system level is not isolated to one specific module. Instead it has impact on many parts of the SW including all interrupt entries, Idle process (erasing is done here), startup part for checking correct termination during last switch off etc.

The achievements/limitations that are done to reduce the RAM requirement and simplify the complexity:

- Interrupt is disabled during write to the flash (disabled for each word write) => no need for interrupt entries in RAM
- Only possible to write to virtual Eeprom (shadow in RAM) during runtime from one process level, which eliminated the need for Eeprom driver to have mail interface, but can be implemented as a clean function library.
- Swap (moving valid data from one block to another, when one block is full) is done within a write command and is not done from back ground process i.e. realistic because of the limited amount of *dynamic parameters*.
- A check during initialization that guarantees always a valid Eeprom even though the dynamic part some how (theoretically) should be damaged.

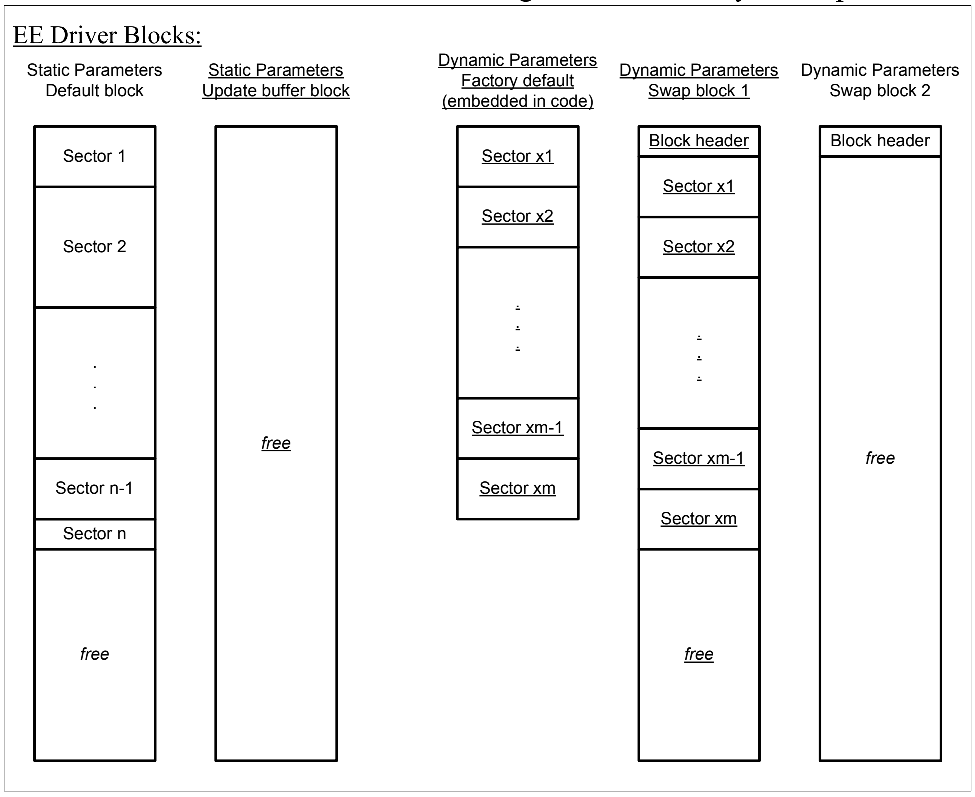
This is the concept that is implemented. This concept supports all the flash-types both the Intel/Micron flash-types which support "write suspend" and the Fujitsu/AMD types which doesn't support "write suspend".

[1] In some flash device it is possible, but they will be considered as not realistic due to fact, that they are to expensive.

2.2 Detailed Description

2.2.1 Eprom structure in flash memory

Below is depicted the blocks in the flash used for storing the static and dynamic parameters:


EE Driver Blocks:
Static Parameters Default block
Static Parameters Update buffer block
Dynamic Parameters Factory default (embedded in code)
Dynamic Parameters Swap block 1
Dynamic Parameters Swap block 2
Sector 1
Sector 2
Sector n-1
Sector n
free
Block header
Sector x1
Sector x2
Sector xm-1
Sector xm

Static Parameters

The *static parameters* are stored in a linear way given from the type definition for the data structure. The *static parameters* are programmed in steps during the production flow so each address is only programmed one time. If some parameters has to be reprogrammed all *static parameters* has to be read and programmed in the update buffer block, then the block must be erased before the *static parameters* again can be programmed.

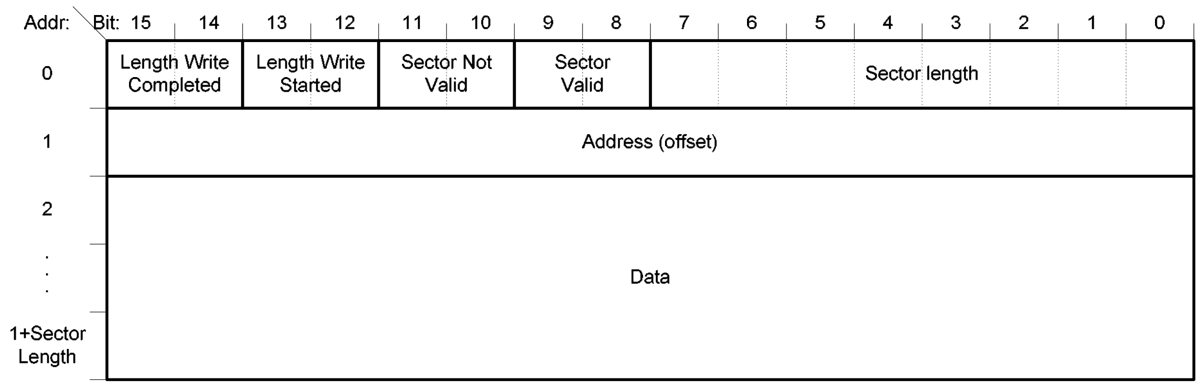
By having the *static parameters* stored as described, bookkeeping is avoided and access to the parameters is made very simple. By having its own block it is also avoided to swap the *static parameters*. In normal operation mode the static block will be locked protected for writing – the only way to unlock it, is by using dedicated functions reserved for the Test equipment and Debug tools.

Dynamic Parameters

The *dynamic parameters* are stored in a segmented way in sectors. Before the software is build, the factory settings of the *dynamic parameters* are included in a source file for the build. The factory default is stored in sectors, which equals the format held in the flash. This formatting of the data is done by a tool (EEP Tool) before building the software. The segmented data image is stored in eep.c by the tool.

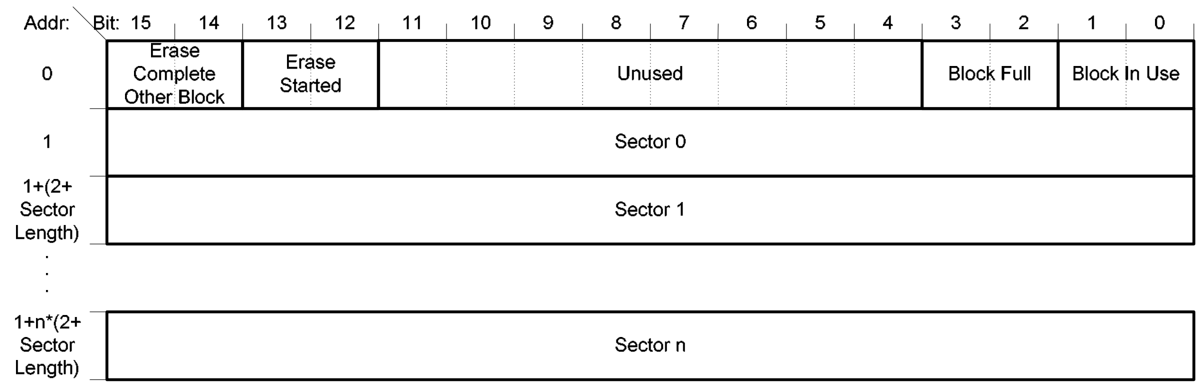
Below is depicted the layout of each sector:

Sector Layout:



Below is depicted the layout of the block header and sectors in the *dynamic parameters* block:

Block Layout:



At power up EE_initialize() is called, and a consistence check is made by comparing the 2 swap blocks with the factory setting block. It will be recognized if the swap blocks are un-programmed or corrupted, and then one of the swap blocks will be initialized with the contents of the factory settings. When writing to the "swap block" also the block header information is initialized. If all the sectors are found in one of the swap blocks, this data is used.

The "swap blocks" are used during runtime in normal operation mode. If a parameter is changed, using the Eeprom driver, the sector is found containing this parameter, and the entire sector is rewritten with the updated parameter on the next free location in the block, which is in use. It is possible to change more bytes at the time and if data is located in different sectors then just all sectors are updated and rewritten.

The image containing the factory setting of the *dynamic parameters* can not be changed during run time and is only used for guarantying the possibility to restore the *dynamic parameters* if both of the "swap blocks" should be corrupted.

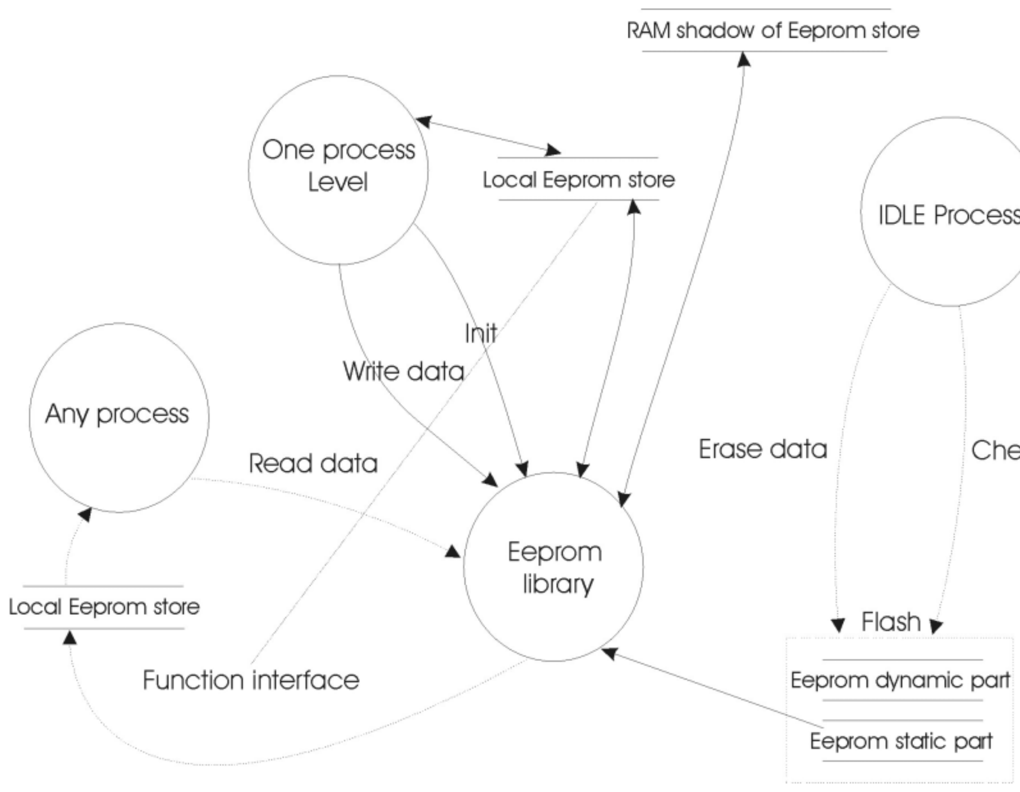
2.2.2 Architecture

The Eeprom emulation SW is implemented as a function library running on the calling process priority. It is possible to make read access from more processes with different process levels (priority), but regarding the write access, it is limited to be none-reentrant and thereby does not support to be called from more than one process level. Anyway there is one restriction to the read access processes: **It is not allowed to make read access from a process level with lower priority than the priority of the writing process.** When reading dynamic parameters data a function is called with destination address, which is an address to a local store in RAM where data should be copied to; source address, which is the offset address related to the dynamic parameters data structure from where data should be read in the Eeprom and the number of bytes to read. The Eeprom library function then copies the requested number of bytes from flash to RAM. Due to the fact that the *dynamic parameters* are stored in a segmented mode it is not possible to read the parameters without using the Eeprom driver.

When reading static Eeprom parameter it can be done using the read function in Eeprom library for *static parameters*, which work identical as when reading *dynamic parameters*, but it is also possible just to read directly from flash e.g. data are never changed and stored in a linear way related to the Eeprom data structure.

When writing dynamic Eeprom data a function is called with source address, which is an address to a local store in RAM where data should be copied from; destination address, which is the offset address related to the Eeprom data structure from where data should be written in the Eeprom; and the number of bytes to write. The Eeprom library function then copies the requested number of bytes from RAM to a copy of the dynamic structure in RAM. Writing to the static part is done using dedicated functions only used in production and in LAB and is excluded in the drawing below

Erasing & writing to the flash is done from IDLE process.

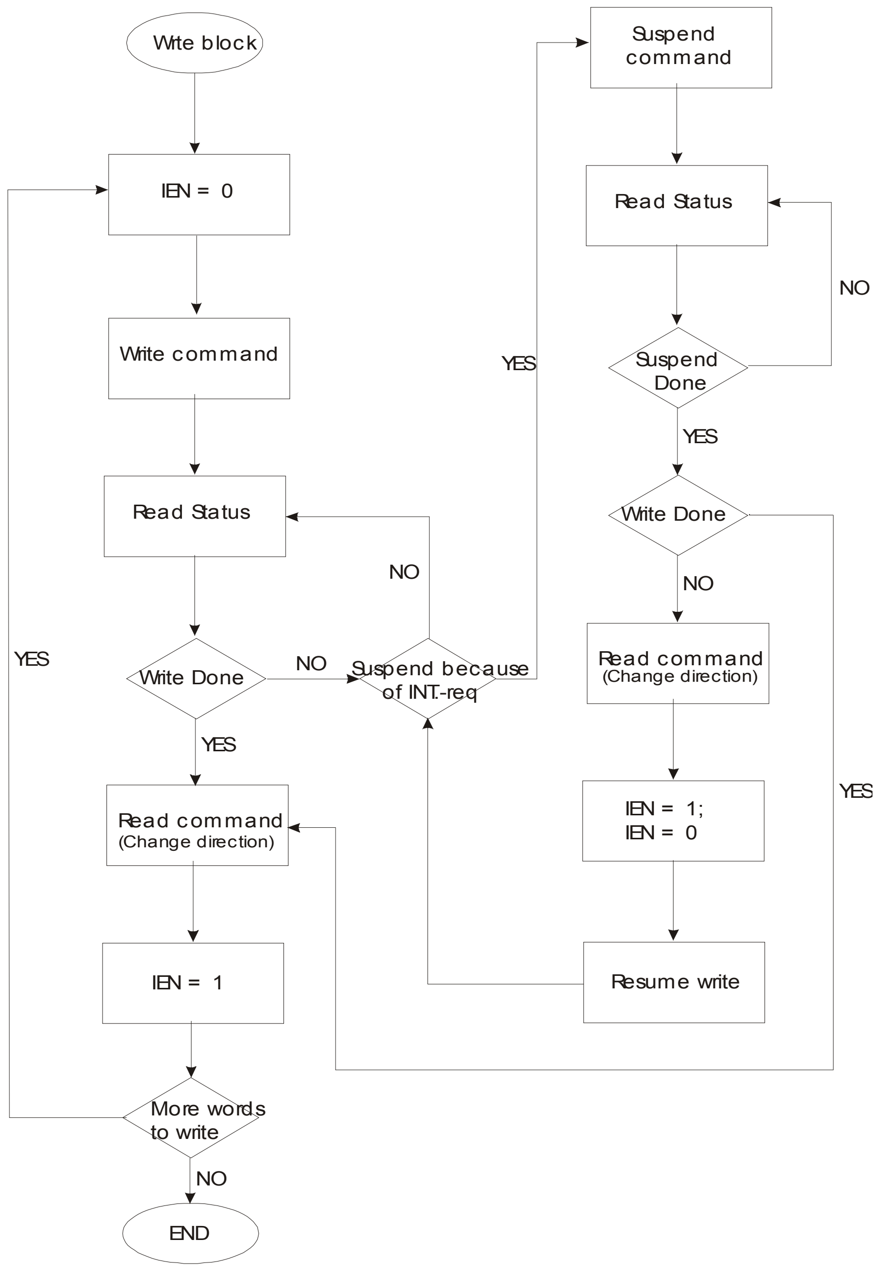


2.2.3 Low Level Write

To avoid all interrupt entries to be located in RAM the implementation of Eeprom emulation is made so interrupt is disabled during write to flash, meaning that only the function which physically write the word to the flash has to be located in RAM.
Writing a sector is split up in a number of iterations and in between interrupts is allowed to be handled by enabling interrupt. Each loop of the iterations will consist of writing one word to the flash.

Intel flash:
During the write of a word, which typical last 22 [µs] but can last up to 200 [µs], a list of “essential” interrupts, which are defined in the way, that they can’t allow to be held for this period of time, are monitored. If one of the “essential” interrupts are requested, suspend is carried out and the interrupt is allowed to be executed. After the interrupt is finished the write is resumed.

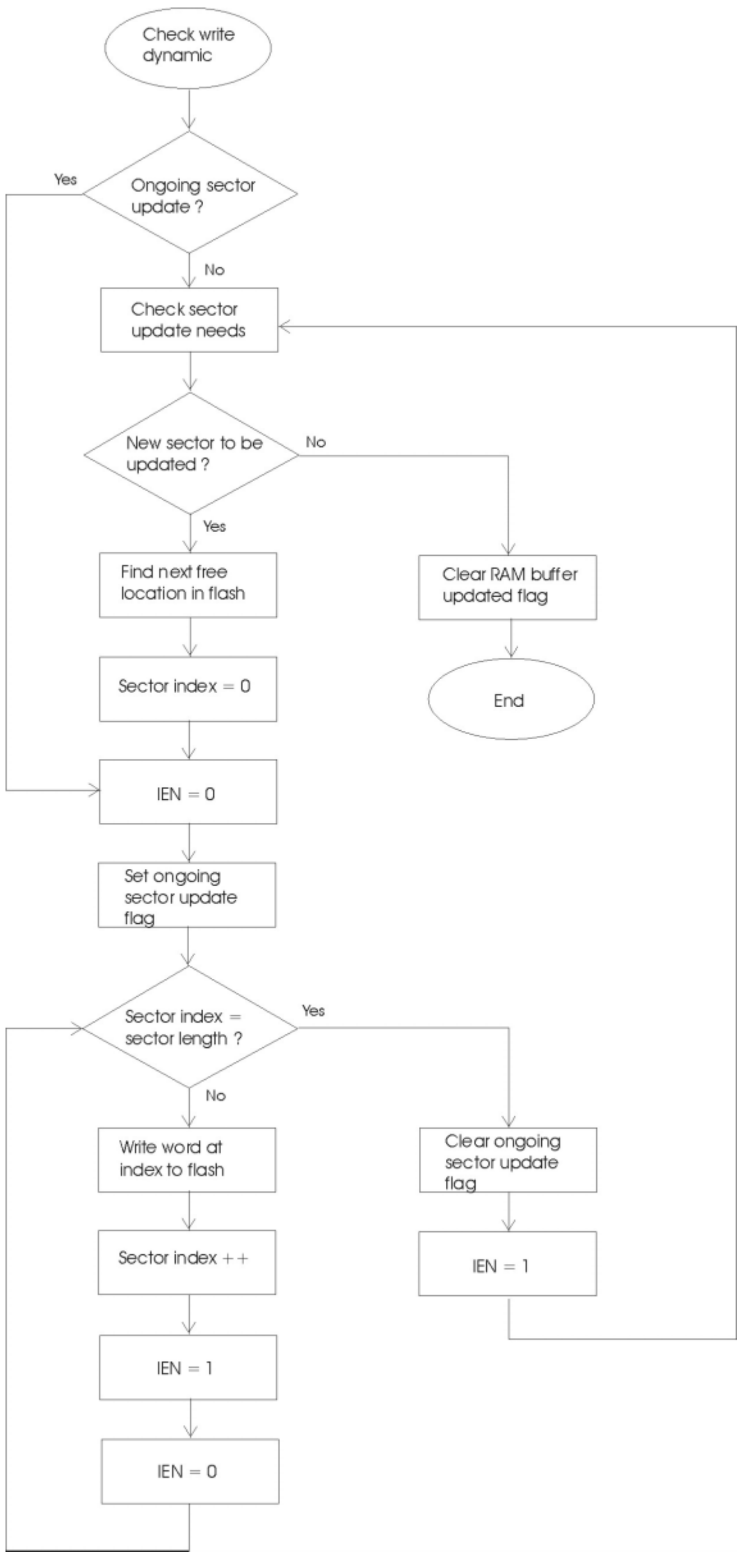
Below is illustrated how a sector, containing header information and sector data is written. There is not distinguished between header fields and sector data e.g. they are all treated logical as a number of words to write before a complete sector is written:



Fujitsu flash:

During the write of a word, which typical last 15 [µs] but can last up to 360 [µs], all interrupts are disabled. Between each word write interrupt is allowed. Write is done from the idle process.

Below is illustrated how the physical Eeprom is updated.


Check write dynamic
Ongoing sector update ?
Yes
No
Check sector update needs
New sector to be updated ?
No
Yes
Find next free location in flash
Clear RAM buffer updated flag
Sector index = 0
End
IEN = 0
Set ongoing sector update flag
Sector index = sector length ?
Yes
No
Write word at index to flash
Clear ongoing sector update flag
Sector index ++
IEN = 1
IEN = 1
IEN = 0

2.2.4 Dynamic Parameters Bookkeeping

On top of the low level write functions, an algorithm is running handling the bookkeeping for the dynamic parameters structure. When updating parameters the first thing is, to find the sector in the Eeprom

containing the parameters that is going to be changed. If not all updated parameters are located in one sector the update procedure is split up in more sector updates.
When writing the updated sector the next free location in the Eeprom has to be found. If there is sufficient space available in the block currently being used, the sector is simply written on the free location found using the Write Sector scheme describe in section 2.2.5, else the "*block full*" bit is cleared in current used block header and the "*Block in Use*" bit is cleared in the block header associated with the other block. It is assumed that the other block has been erased, and all valid sectors are hereafter copied from the old block to the new block and the "*Erase Started*" bit is cleared in the old block header indicating to the IDLE process, that erase is needed, when the erase has finished, the "*Erase Complete Other Block*" bit is cleared in the new block, which then holds all dynamic parameters..

2.2.5 Write Sector

Updating a sector need to be power loss proof in such a way that either the new or the old data can always be recovered. Below is given the procedure that is used to write sectors.

1. Clear "*Length Write Started*" bit in the sector header.
2. Program sector length in the "*Sector Length*" field in the sector header, e.g. the number of words in the sector excl. sector header.
3. Clear "*Length Write Completed*" bit in the sector header, indicating that the length is programmed OK
4. Program "*Address*" field in the sector header, e.g. the offset address related to the beginning of the dynamic Eeprom data structure.
5. Program all sector data
6. Clear the "*Sector Valid*" bit in the sector header, indicating that the data has been written correctly.
7. Clear the "*Sector Not Valid*" bit in the sector header of the old sector, to indicate that this sector no longer is valid
8. Done

If the power is lost during any of the steps, it will always be possible to recover by analyzing the sector header bits, in order to find where the power was lost, and which counter-actions must be taken. Either the new sector or the old sector will be valid. If the new is not valid, the bit "*Sector Not Valid*" will be cleared during initialization of the system. If the "*Length Write Completed*" bit is not cleared, the length field can not be trusted, so it is set to zero together with all the other bits in the header (except for the "*Address*" field, which has not been written).

2.2.6 Initialization

During the power up procedure of the MS an initialization function of the Eeprom driver is called.

This function handles a copy-function that copies the RAM located part of the Eeprom driver from flash to external RAM.

The consistency of the dynamic block is then checked. If the consistence check fails the dynamic parameters is repaired, if possible, or re-initialized from the factory block.

In the initialization function it is also checking, if the MS was switch off in the middle of a swap procedure and if so the swap is resumed and finished.

The RAM-copy of the dynamic parameters is initialized with the contents of the dynamic parameters from flash.

2.2.6.1 Consistence check

During consistence check of the *dynamic parameters* the current active swap block is first identified. If no active block could be fund the dynamic parameters is initialized from the factory block.
The Eeprom write sector concept is made so it has no influence if the MS was in the middle of a sector update, when it was switched off i.e. controlled by the control bit in sector header. Anyway there are two special situations:
1. If a sector is updated and set valid but the MS is switched off before the invalid bit is set in the previous valid sector. In this case there will be two valid sectors, where it should only be the last one, which is valid.
2. If a sector update is started and the "*Length Write Started*" bit is cleared and the "*Length Write Completed*" bit is not cleared the sector length might be invalid. The sector length should then be cleared to '0'.
The consistence check will correct these special situations. The consistence check will also check that all sectors in the block containing the factory setting if the *dynamic parameters* can be found in the active "swap block" – if not the dynamic part is corrupted and both "swap blocks" will be erased and the factory block will be copied into one of the "swap blocks". This is only implemented as a clean up for a theoretically crash problem, which should never occur. It is also checked if an unfinished block erase has been started, and the erase will be completed (this is done after the swap resume check).

2.2.6.2 Check for unfinished sector update

It might happen that the MS is accidentally turned off in the middle of a sector-update. This situation is recognized if anything is written in the length field and the sector valid bit is not cleared.
Solution: if the sector length is written the sector valid bit and the sector not valid bit is cleared. Otherwise the sector length is set to zero and the sector valid bit and the sector not valid bit is cleared.

2.2.6.3 Check for unfinished block erase

If the MS is turned of in the middle of a block erase, the block may be corrupted and needs to be erased before it can be used as swap block (this is done after the swap resume check). This is detected at startup by noting that the "*Erase Started*" bit is cleared, but the "*Erase Complete Other Block*" is not.

2.2.6.4 Swap, swap resume

Swap resume is identical to **Swap**, which is carried out when a block becomes full during write.
The condition that tells that swap procedure was ongoing during the MS was switched off is: "*Block Full*" bit is cleared in one of the blocks and the "*Erase Started*" bit is NOT cleared.

2.2.7 Read

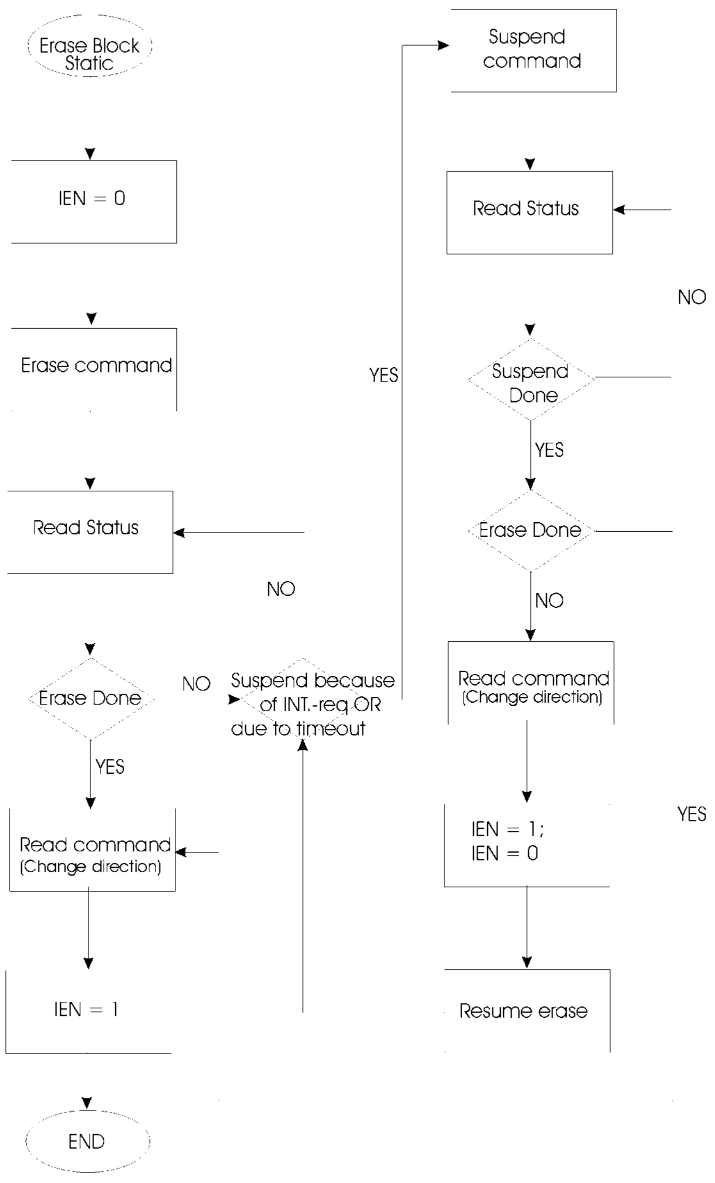
Reading from the Eeprom is very simple compared to write. Reading *static parameters* can either be done through the Eeprom driver interface or directly by accessing the data structure in flash. If the Eeprom driver interface is used the driver copies the wanted amount of *static parameters* from the data structure in flash to the destination address given as parameter in the read-function.
When reading *dynamic parameters* it has to be done through the Eeprom driver due to the segmented way the data are stored.

2.2.8 Erase

Erase is done on block basis and the same concept is used, as when writing to Eeprom. During erase, a list of "essential" interrupts, which are defined in the way, that they can't allow to be held for this period of time, are monitored. If one of the "essential" interrupts are requested, erase suspend is carried out and the interrupt is allowed to be executed. After the interrupt is finished the erase is resumed.
The erase method is slightly different for the static and the dynamic blocks.

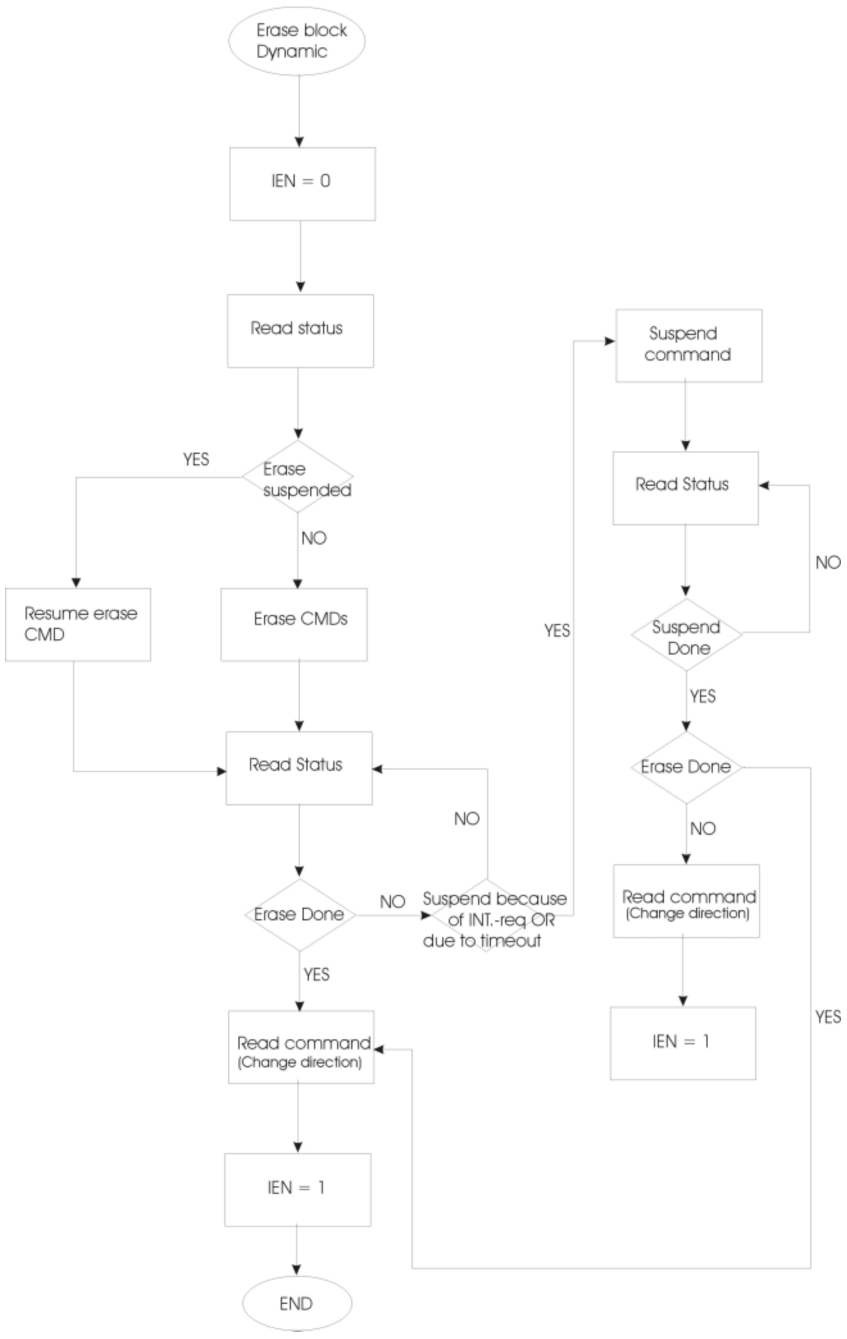
2.2.8.1 Erase static

Below is illustrated how the static block is erased.



2.2.8.2 Erase dynamic

Erase of the dynamic blocks is done from the IDLE process. Every time entering the IDLE process an erase function is called in the Eeprom driver. If there is no need for erasing data the function returns immediately else erase is carried out and the function returns after erasing has been done.
Below is illustrated how a block is erased



Note: Be aware if nested suspend is supported by the flash type (it is for Intel). It is needed if erase is suspended and a write to flash is carried out which again has to be suspended!

2.3 Exception Store

To be able to analyze and debug the run-time behavior of the MS it is useful to have a log containing the exceptions that may arise.
The exceptions should be stored in non-volatile memory for later readout.
For storing exception data a separate block is used.

Each exception will occupy a sector in the block. A sector is consisting of a header containing an exception number and a structure containing the exception data. The header is used to determine the number of stored exceptions in the block (useful during readout or to find the next free location when a new exception has to be stored).

There are two types of exceptions: hardware traps generated by the CPU and software exceptions. The two types is stored in the same store, but are treated slightly different.
The HW traps are requesting an interrupt to handle the trap. From this interrupt service routine the trap information should be stored in a new sector in the block.
The SW exceptions are temporarily stored in RAM, and will be stored in flash if a trap occurs or when the MS is powered off.

The following routines are needed for the exception store:

Store:

Stores an exception in a new sector in the flash.

Read:

Used for readout. Returns the sector with the number specified by an input parameter.

Find next free location:

Locates the next free location in the block when a new exception has to be stored.

Get number of stored exceptions:

Used in connection with read to determine how many sectors to read out.

Check for un-stored exceptions:

Is checking the ram-buffer for un-stored exceptions and is storing the exceptions (if any) to flash.

2.4 IMEI / SP lock.

The IMEI and the SP lock codes have to be stored non-volatile in the MS.
For that purpose the last 256 bytes in the static block is reserved. This 256 bytes sector can only be written via a dedicated function, so it's not possible to write to this sector via the normal function set for handling the static eeprom.
The format of the data in this block is unknown for the EE-driver.

When the static block is updated this sector shall be copied to the static buffer block and back again together with the modified contents of the static block to avoid loosing the IMEI and SP lock codes.