Module specification Flash File System

Subject: Design and interface spec.

Revision: 2.5
Author: Erik Christensen
Last Updated: 09/01/2006 10:50
Last Updated By: Charlotte Hansen
Printout Date: 14/02/2006 03:06
File: C:\TEMP\FFS spec\FFS_specification.doc

History

Revision 0.3	• File header format changed.
Revision 0.4	• Chapters re-arranged. • New error codes: FFS_INTERNAL_FATAL_ERROR, FFS_USER_ID_ERROR and FFS_WRITE_ERROR have been added. • Chapter added: 3.2.5.4 Protection of the RAM based tables. • Chapter rewritten: 3.2.1 Architecture. • Chapter modified: 3.2.2. Mail Interface. • Chapter added: 3.2.5.6 User Id Table. • Chapter modified: 3.2.8.3 Pseudo task operations. • Chapter modified: 3.3 Interface description to the FFS driver. • Chapter rewritten: 3.1 Definition of concept.
Revision 0.5	• Minor changes in the function-prototypes in the “Interface description to the FFS driver” chapter. • FFS_MEMORY_RES_ERROR added to the error codes. • FFS_MAX_NOF_FILES_EXCEEDED added to the error codes. • FFS_MAX_NOF_OPEN_FILES_EXCEEDED added to the error codes. • Minor changes to chapter: “Pseudo task operations”. • Minor changes to chapter: “Architechture”.
Revision 0.6	• New error codes has been added. • Minor changes in the function-prototypes in the “Interface description to the FFS driver” chapter.
Revision 0.7	• Function FFS_get_id () added. • FFS_REQUESTED_MEM_NOT_AVAILABLE added to the error-codes. • FFS_get_file_info() changed. • Error code table updated.
Revision 1.0	• New template used. • New chapter added: “Physical file structure”. • Max. number of filetypes changed.
Revision 1.1	• New functions FFS_findfirst() and FFS_findnext() has replaced FFS_get_id() and FFS_get_id_b y_type().
Revision 1.2	• The functions FFS_findfirst() and FFS_findnext() description changed. • Added functions to use the FFS with filenames. • Added callback parameter in FFS_close, FFS_create, FFS_write, FFS_read, FFS_append, FFS_modify, FFS_delete, FFS_rename. • Added user_id parameter in FFS_open • Swapped the *dst and offset parameters in FFS_read to align it with FFS_write and FFS_modify • Added FFS_FALSE, FFS_TRUE, FFS_INVALID_FILENAME to the error code table. • Removed ‘type’ field from the sector header.
Revision 1.3	• A linear file access method is added. To support this method the functions FFS_get_ref_LFA(), FFS_load_to_LFA_FN() and FFS_load_to_LFA() has been added.
Revision 1.4	• Added a split of the FFS into a static and a dynamic area. • Added factory default file concept. • Added file identification section. • Updated the system architecture section with asynchronous interface mode • Updated the GC section. • Added Out of Memory Error Message at Special File Operations • Added FFS LFA and the Task Erase Issue
Revision 1.5	• Added Truncate and copy in File operations.

Revision 1.6	• Added section with compiler define descriptions. • Updated the module content with the new FFS source code files. • Added a chapter listing the PC tools, which can be used with FFS. • Added description of the FFS Management tables: FFS_blocks, Sector usage and HW block no. • Changed memory header figures to have address 0 at the top and updated the headers. • Added some more examples of memory overhead calculations • Added Truncate and copy pseudo task descriptions. • Updated the other pseudo task descriptions. • Updated the description of the garbage collector. • Updated the description of the Power Loss Recovery functions • Updated the configuration at power up with the new memory tables. • Added a description of the bank switch concepts • Added a description on how to customize FFS with the ffs_def.h file • Added a FFS version table
Revision 1.7	• Directory support added. • File-header changed
Revision 1.8	• Chapter "FFS structure in flash memory" updated.
Revision 1.9	• Added Restore all factory files function
Revision 2.0	• Rewrote the FFS Memory section
Revision 2.1	• Forced GC section added
Revision 2.2	• Added wear leveling documentation and test results in appendix • Added GC and forced FFS state machines • Updated Garbage Collection chapter
Revision 2.3	• Limitation chapter updated
Revision 2.4	• Streaming chapter updated with synchronous streaming mode.
Revision 2.5	• Added a segmented region FFS update of the FFS description. • Added a buffered write description.

Terminology

Glossary

Abbreviations

Contents

1 Introduction

1.1 Overview

In a Mobile Station (MS) there is a need to store data in a non-volatile memory, so the data contents is not lost when the MS is switched off. The types of data to be stored non-volatile can be split up in the following groups:

1. Adjustment parameters used to calibrate the MS during production. These parameters have a fixed size and structure.
2. Parameters that can be modified during run-time in normal mode (e.g. MMI settings). These parameters have a fixed size and structure.
3. Application data where size and structure are unknown at compile-time.
4. Application data that will be created/deleted dynamically at run-time.

NV data of type 1 and 2 will be handled by an EEPROM emulation driver, and is beyond the scope of this document.

NV data of type 3 and 4 is to be handled by a Flash File System (FFS), which is described in this document.

Examples of items that could be stored in the FFS:

- SMS
- Phonebook
- Calendar
- E-mail
- Address book
- WAP profiles
- Ringing tones
- Bitmaps
- Animations

1.2 Features

- Handling of multiple files in a mobile station
- Simple function-interface towards the application level.
- Mechanisms to handle multiple files opened for read/write simultaneously.
- Mechanisms to handle multiple files opened for streaming read/write simultaneously.
- Mechanisms to handle multiple processes with simultaneous FFS-access.
- Files can be predefined at compile-time, or be created/deleted dynamically at run-time
- Files are grouped into static files and dynamic files. The static files are protected against been changed on runtime, they can only be created by external tools.
- Factory default files. Static files can be created as factory default files and FFS will then ensure there is a duplicate dynamic file.
- Linear Flash Area. Files in FFS can be loaded to a linear accessible flash area.
- Power Loss Recovery algorithms to prevent corruption after unintended power loss.Garbage Collector reclaims space marked as invalid by the different FFS operations.
- Support of the following memory medias:
  - Single bank flash types (combined code-storage and FFS data in the same device).
  - Dual bank flash types (combined code-storage and FFS data in the same device).
  - Separate parallel data flash types.
  - Can be adapted to serial data flash types, Smart Media cards etc.
  - Segmented Region flash types.

1.3 FFS Versions

FFS has experienced some shifts in version. The version is shifted when the header design has been changed. This design change makes it necessary to make new FFS files if it is wanted to make files with the image creator tool to download with the rest of the software for the mobile.

VERSION	CHANGE at FFS
…	…
8	The possibility to use factory default files added linked default file ID at the file header. With the inclusion of the LFA facility, it was also decided to add a word more for security signature in the block header.
9	Use of logic block numbers added LBN at the block header. This version added another GC and allocation of sectors where the hw_block_no table is used
10	All block headers have been changed to support multi level flash handle. This made it necessary to use 2 bits for status bits. In addition, DRM attributes local id and mime type has been added at the file header. The security signature in the file header has been removed.
11	Directory support. The File-header has been changed.

VERSION	CHANGE at FFS
12	Support of Segmented Region flash types. The FFS has been redesigned to handle the Segmented Region flash type (like the Sibley flash). The version number has been added the extra security signature in the block header. This makes FFS clean a FFS image that doesn't fit the image.

1.4 Limitations

- Max. number of files: 0-65536 (configurable i.e. 1200).
- Max. number of filetypes: 0-256 (configurable).
- Max. file size: 4 Gbytes.
- Max. size FFS storage: 4 Gbytes.
- Max. filename size: 48 chars (47+\0)
- Max pathname size (including filename): 128 chars (127+\0)

1.5 Compiler defines

1.5.1 Feature defines

Define	Description
FFS_PRESENT	Includes the entire FFS.
FFS_FN	Includes the filename interface feature.
FFS_LFA	Includes the Linear Flash Area feature.
FFS_FACTORY_FILES	Includes the Factory default files feature.
FFS_STREAMING_PRESENT	Includes the Streaming interface feature.
FFS_BANK_SWITCH	Must be set if any of the bank switching concepts are used
FFS_BANK_SWITCH_CS0	Use the CS0 bank switch concept
FFS_BANK_SWITCH_CS2	Use the CS2 bank switch concept
FFS_SET_BANK_SWITCH	Macro to perform the bank switch, this must be defined if using bank switch.
FFS_RESET_BANK_SWITCH	Macro to switch back from the bank switched mode, this must be defined if using bank switch.
TRAP_HANDLING	If defined FFS will issue traps on fatal errors and store exceptions on non fatal errors

1.5.2 Platform defines

Define	Description
SGOLD	Must be set when using the SGOLD chip
SGOLDLITE	Must be set when using the SGOLDLITE chip
DWD_HW_P2002_PLUS_ADLER	Must be set when compiled to the ADLER SW
_NUCLEUSTYPES_H	Must be set when using the NUCLEUS OS
_OSE_TYPES_H	Must be set when using the OSE OS
_16MBIT_FLASH	The flash have a size of 16 Mbit
_32MBIT_FLASH	The flash have a size of 32 Mbit
_64MBIT_FLASH	The flash have a size of 64 Mbit
_96MBIT_FLASH	The flash have a size of 98 Mbit
_128MBIT_FLASH	The flash have a size of 128 Mbit
_256MBIT_FLASH	The flash have a size of 256 Mbit
GTB_SUPPORT	Must be set when compiled to the GenericTestBench SW
EE_MULTI_LEVEL_FLASH	This is a special feature of a flash utilizing 2-bit-per-cell technology which increases the size of a 64 Kbytes block to 128 Kbytes
EE_BUFFER_PROGRAM	The flash is able to utilize buffered programming
EE_SIBLEY_FLASH	If defined FFS will use write modify writing to the flash (to avoid rewrites) and the data writes will be aligned with the regions supported by the write interface to the flash. In addition, write buffering will be used.

1.5.3 Test defines

Define	Description
DWD_MODULE_HOST_TEST	Enables FFS to run in a host environment on a PC. This is mainly used to test FFS.
FFS_PERFORMANCE_MEASUREMENT	Includes functionality used to make performance measurements of FFS

1.6 Module contents

File	Description
FFS_INTERFACE.C	Contains all the source code to the FFS interface function calls
FFS.C	Contains all the source code for the core functionality of FFS
FFS_DIR.C	Contains all the source code for the directory FFS functionality.
FFS.H	Specifies the interface to the FFS.
FFS_DEF.H	Contains configuration settings to customize the FFS by including FFS_DEF_xxtype.h (FFS_DEF_TYAX.H, FFS_DEF_SIBLEY.H or FFS_DEF_TYAX_2x256.H).
FFS_OPCODES.H	Specifies all the AT# commands understood by FFS
FFS_INTERNAL.H	Specifies all variables shared between the FFS files.
FFS_TYPES.H	Defines the variable types used in the FFS code
FFS_MONITORING _MESSAGES.H	Contains interface to FFS Mobile Analyzer decoding.

1.7 FFS tools

Since FFS is used to store data files there is made a special interface to FFS and some dedicated tools to make it possible to test FFS and exchange files between FFS and a PC.

The special interface uses AT# commands and is described in reference [3]. It has a command to access nearly all the normal FFS interface functions.

FFS is tested with the GenericModuleTestSystem (described in reference [5]) through this interface.

The files in FFS can be exchanged with a PC by using the PhoneTool, which also uses this interface.

If there is a lot of files which shall be preinstalled/downloaded to FFS it can be more efficient to use the Storage Tool (described in reference [4]) to create an image of the entire FFS data storage area and download this image with the FlashTool (described in reference [6]).

1.8 A Segmented Region FFS

The FFS must be able to run at a segmented region type flash like the Sibley flash. This means that the FFS structure must adapt to the limitations that this flash type gives. The limitations for the Sibley flash are described here against the features for a typical Tyax flash, the Intel Strata flash.

Strata flash	Sibley flash
▪ Single word programming possible ▪ Buffered programming possible ▪ Bit twiddling possible	Each block is divided into programming regions of 512B/1KB each. Each region is divided into 32 segments of 32 bytes. A region can be configured for two different modes (object or control mode): Object mode: ▪ Region size: 1KB ▪ The segments can be used totally.

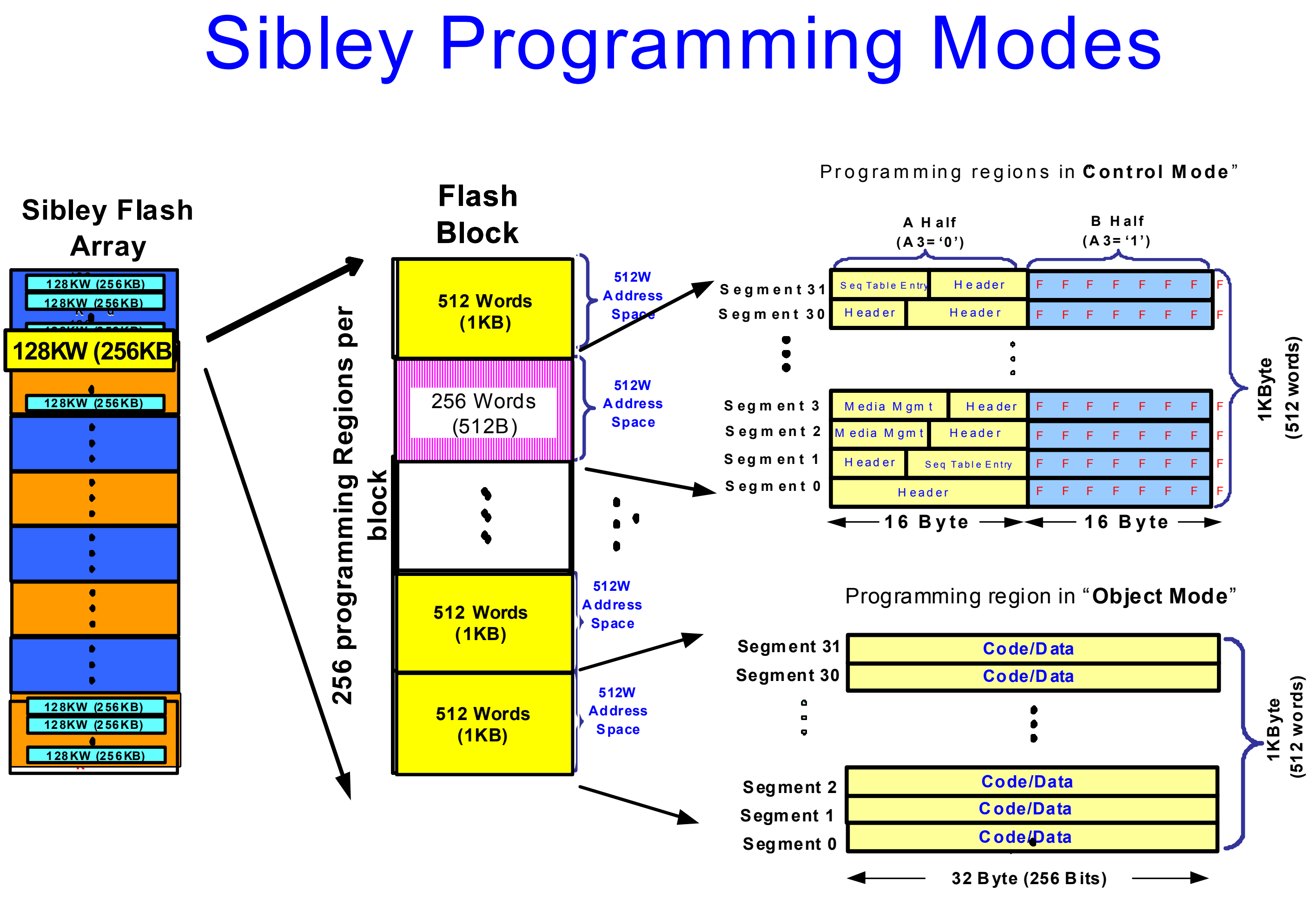
	▪ Buffered programming only. The region can be programmed only once before erase. ▪ Single word programming is not possible. ▪ Bit twiddling is not possible Control mode: ▪ Region size: 512B (half of the memory space is wasted in control mode) ▪ Half of the segments can be programmed. The last 16 bytes must be 1's. ▪ Single word programming possible ▪ Buffered programming possible ▪ Bit twiddling possible

The Sibley flash design is illustrated below. The picture is taken from the "Comneon Sibley Update June 2005 PowerPoint".

The programming distinguishing between control mode and object mode makes it necessary for the FFS to program

- All headers in control mode (segmented)
- All data references in control mode (segmented)
- All data in object mode (program in region sizes once before erase)

The sector size in FFS must be the same as the Sibley region size.

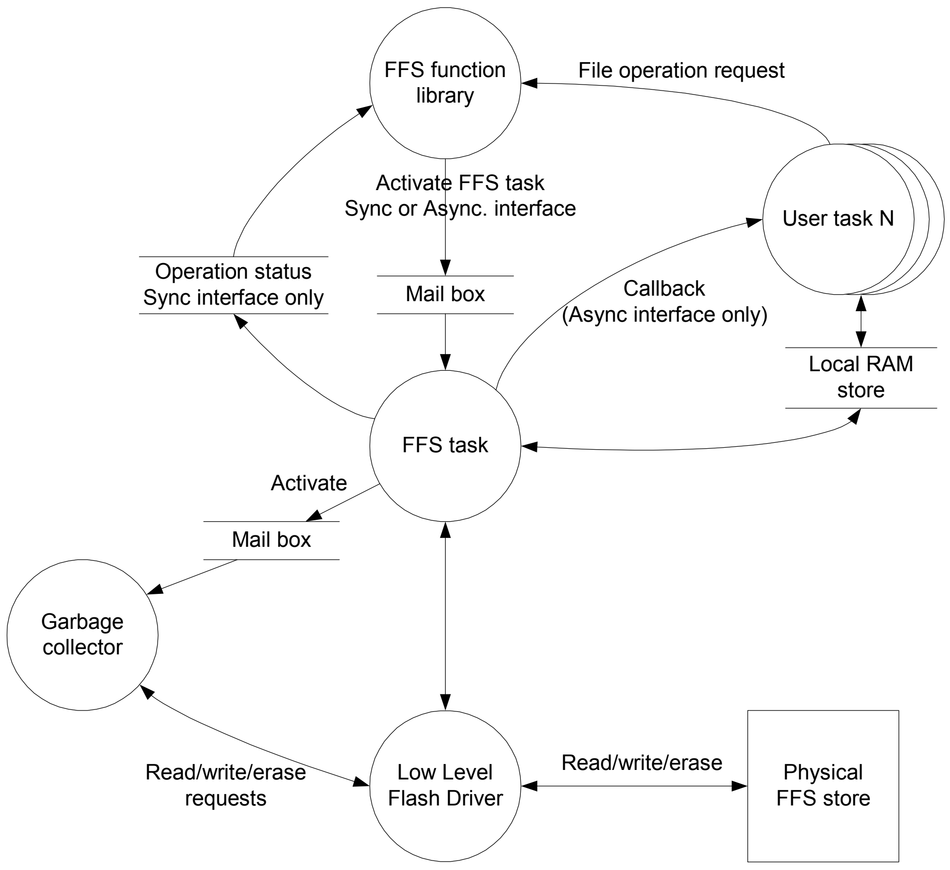


2 Architecture and functionality description

In the following chapter there will be a specification of the FFS architecture and functionality.

2.1 System Architecture.

An overview of the Flash File System is illustrated below:



From the application point of view, the interface to the FFS is a function library. All functions can be called in a synchronous mode where they don’t return before the requested operation is completed. This eliminates state-handling and synchronization problems on the application level. As an option, the application can select to call the functions in an asynchronous mode, where the functions return immediately after validation of the input parameters and the result of the operation is returned with a callback function.

From the application point of view, both read and write operations can be done reentrant from any process level, and several files can be simultaneously open for read/write-operations, but the individual files can only be opened in one instance.

Physically only one FFS-operation can access the flash at a given time. Flash access is done from a separate FFS task with a mail-interface. All mails have a priority based on an FFS priority assigned to the calling process. An operation requested from a process with a low FFS priority can then be interrupted by an operation requested from a process with a higher FFS priority.

When files are deleted or modified then the FFS operations result in some parts of the flash are marked as obsolete. A garbage collection is needed to reclaim these obsolete parts of the flash. This garbage collection is done from the garbage task.

The **Write Operations** below is a category of all the operations that physically changes the flash:

- **Create**
- **Write**
- **Modify**
- **Append**
- **Close (write, append, modify)**
- **Delete**
- **Rename**
- **Load (to LFA)**
- **Copy file**
- **Restore factory default file**
- **Truncate**
- **Set DRM attributes**

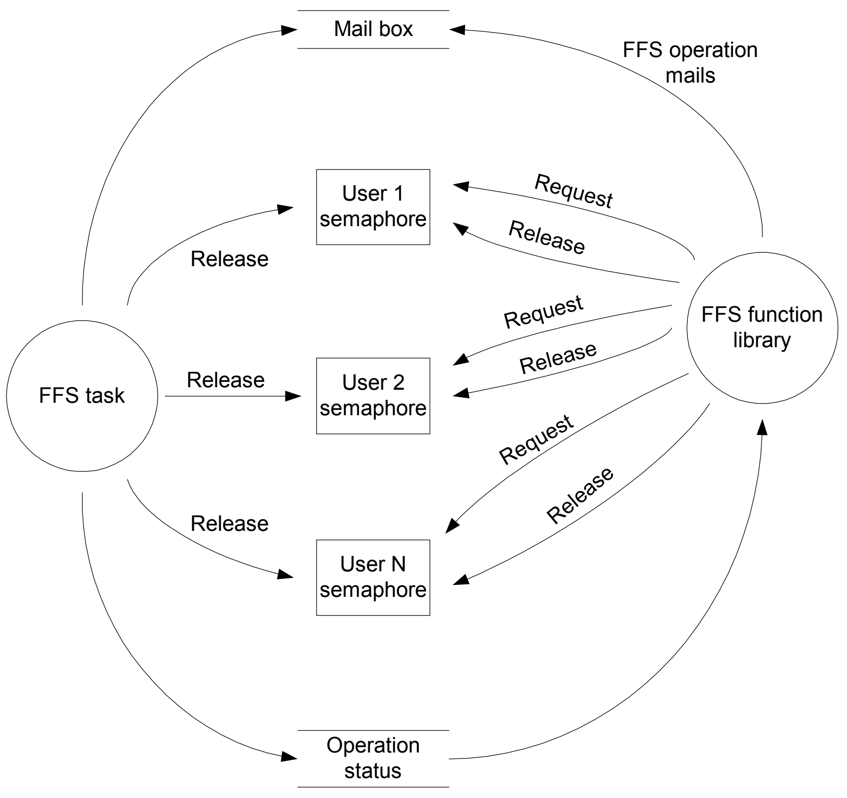
The **Read Operations** below is a category of all the operations that doesn't change the flash:

- **Open**
- **Close (read)**
- **Read**
- **Get file info**
- **Get (LFA info or reference)**
- **Find files**

These operations will be handled by the FFS task.
The status information (error codes) for these operations is made available for the function interface by using the Operation Status data-structure containing a status field for each user.

2.2 Mail Interface.

The mail interface for the FFS is illustrated below where semaphore requests and releases are used only with synchronic FFS operations.



All FFS operations that access the flash are delivered as mails to the FFS mailbox via the function interface. The mailbox should have space for N mails, where N is the number of FFS-users in the system, since a user (process) only can have one outstanding mail at a time.

The FFS interface data will have the following format:

User ID
Operation Type
Operation Data

The **User ID** is the id of the process requesting the operation. The FFS task can find the corresponding FFS priority in the User Id Table.
The **Operation Type** specifies the type of operation requested (write, create, save, delete, append, rename, …).
The **Operation Data** contains the parameters from the original function call to the function interface.
The **Operation Data** structure is a simple byte array to avoid having different mail types for the different operations.

2.2.1 The FFS interface modes.

The FFS interface can be in synchronized or in asynchronized mode. Even the asynchronized mode is partly synchronized because one user can only have one outstanding operation request. To achieve 'synchronization' between the FFS task performing the operations and the function interface requesting the operations a flag per user task is used to indicate if the user has an active operation.

To achieve synchronization between the FFS task performing the operations and the function interface requesting the operations a semaphore per user task is used.

The first interface steps are similar for synchronized and asynchronized mode.

1. An FFS function is called from a user task x.
2. The FFS function checks if the user already has an active operation.
3. If any of the input parameters is invalid or the user already has an active operation the FFS function returns with an error code.

The last interface steps are depending on the interface mode.

Synchronized interface mode:

4. The FFS function requests semaphore x
5. The FFS function sends the operation-request via mail to the FFS task.
6. The FFS function requests semaphore x again and there by setting the user task to wait.
7. The FFS task carries out the operation.
8. The FFS task copies the result code of the operation to the "Operation Status Buffer".
9. The FFS task releases semaphore x to release the user task.
10. The FFS functions second semaphore request is granted.
11. The FFS function releases semaphore x again.
12. The FFS function reads the result code of the operation from the "Operation Status Buffer".
13. The FFS function returns the result.

Asynchronized mode:

4. The FFS function sends the operation-request via mail to the FFS task.
5. The FFS function returns success.
6. The FFS task carries out the operation.
7. The FFS task calls the callback function with the result of the FFS operation.

2.3 File identification

Each file in FFS is identified by an id. As an option, they can also be identified with a filename. This requires that they are created with a filename.

When a file is created with a filename, FFS generates an id, stores the name as the first bytes of data and marks the file as a file with a name. These bytes of data are hidden from the user. They can't be read with the normal read functions and they are not included in the file size.

A file with a filename can still be accessed with its id.

2.4 File-Handles.

To control the access to files in the FFS and to prevent more processes to operate on a file simultaneously, file handles are used.

When requesting the *Create* and *Open* operations a new file handle is created.

Physically the file-handle is a pointer to the start sector field in the FAT. The file-handle is returned to the requesting user of the file.

When requesting read/write operations the file *id/name* and *handle* is given as input parameters in the function call. This means that only the process holding the *handle* is allowed to operate on the file.

When using both *id/name* and *handle* as input parameters, it is possible to check that the *handle* is matching the *id/name* as an extra security.

2.5 File operations

2.5.1 Create file.

New files can be created in the dynamic part by this operation. When file creation is successfully performed, data can be written to the file.

2.5.2 Open.

Opens a file in one of the following modes:

- Read mode
- Append mode
- Modify mode
- Streaming read mode
- DRM Attributes mode

When opened in a specific mode, only operations allowed for the specified mode are possible.

2.5.3 Close.

Closes a file opened in read/write/append/modify/DRM attributes mode.

2.5.4 Read.

When a file is opened in read mode read access is allowed.
Reads a block of data from a specified location in the file to a specified RAM location.
This operation is implemented reentrant, and multiple files can be open for read simultaneously.

2.5.5 Write.

After file-create the new file is opened for write. Data can be written to the file until the file size is reached or until the *Close* operation is carried out

Writes a block of data from a specified RAM location to a specified location in the file.

Multiple files can be open for write simultaneously.

2.5.6 Modify.

In modify-mode an existing file can be modified by this operation. The whole file or a part of the file can be modified. Note: This operation is very time consuming and should only be used for relatively small files.

2.5.7 Append.

An existing file can be appended by reserving an extra memory area for new data.
The file has to be opened in append mode.

2.5.8 Truncate.

An existing file can be truncated to a new size. The truncate operation will discard data from the end of the file. The file should be in the closed state when requesting this operation.

2.5.9 Delete.

A file can be marked as deleted, and will be cleaned up by the garbage collector in the GC task
The file should be in the closed state when requesting this operation.

2.5.10 Copy.

A new file, which is a copy of another file, can be created. The file should be in the closed state when requesting this operation.

2.5.11 Rename.

An existing file with a specified *Id/Name* can be renamed to another *Id/Name* and/or be converted to another file-type.
Is useful when modifying an existing file by using a temporary file.
The file should be in the closed state when requesting this operation.
The new *Id/Name* must not be used by any other file.

2.5.12 DRM Attributes.

The file header includes the DRM Attributes settings: local id and mime type. They have the default setting 0 which is set when a file is created. With Set_drm_attributes() it is possible to set the DRM attributes to other values. The file should be opened in DRM Attributes mode when requesting this operation.

2.5.13 Restore factory default file.

To support factory default files, see 2.8.1, a static file can be set as a default for a dynamic file.

2.6 Streaming read / write

Streaming access normally uses the asynchronised interface mode to read and write functions. However it's also possible to use the synchronously interface in streaming.

Files used for streaming access must be created or opened in a special streaming mode.

Streaming files can be opened/created from one user and used by another user.
This means a 'control' user (E.g. MMI) can open a file and hand the filehandle to another user (E.g. mp3 player). This user can then stream small blocks of data to or from the file. The MMI can at anytime stop the streaming user and close the file. It shall be emphasized that the controlling module and the streaming user can be the same user.

The streaming user must operate with at least two buffers; one with data the streaming user can operate on while FFS is filling another with data or storing data from.

To make the streaming user independent of the storage media, a streaming storage abstraction layer (SSAL) is introduced. The controlling user opening/creating the file must use a SSAL user id matching the streaming users SSAL user id and tell the streaming user which SSAL media type to use. With this SSAL media type can the SSAL forward the read/write request to the right storage media.

To improve the performance of streaming data the streaming operations are done by a separate process, which can have a higher priority than the normal FFS process.

The Figure 2.1 shows an example where the MMI wants to play a MP3 song from FFS. As can be seen on the figure MMI is the controlling module, FFS is the data providing module and the MP3 module is the data consuming module.

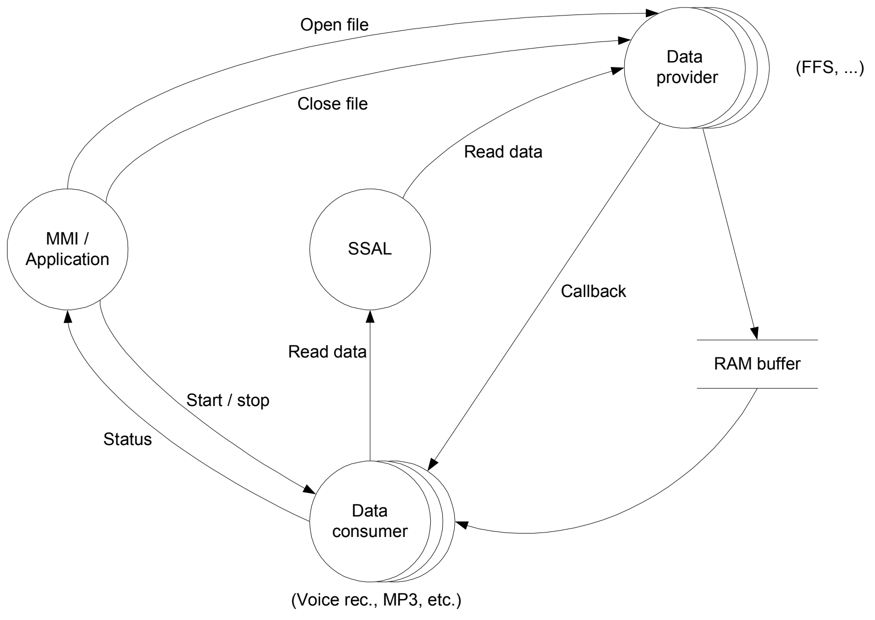


Figure 2.1 Concept for streaming data from FFS

The concept for writing data is identical to reading data from FFS. The only difference is that FFS now becomes the data consuming module. On Figure 2.2 the concepts for writing a file to FFS is illustrated by an example, where the controlling module, the MMI, wants to write a picture to FFS. The camera driver is the data providing module. When the last byte has been written to FFS, the camera driver will inform the MMI, to indicate close the file in data consumer.

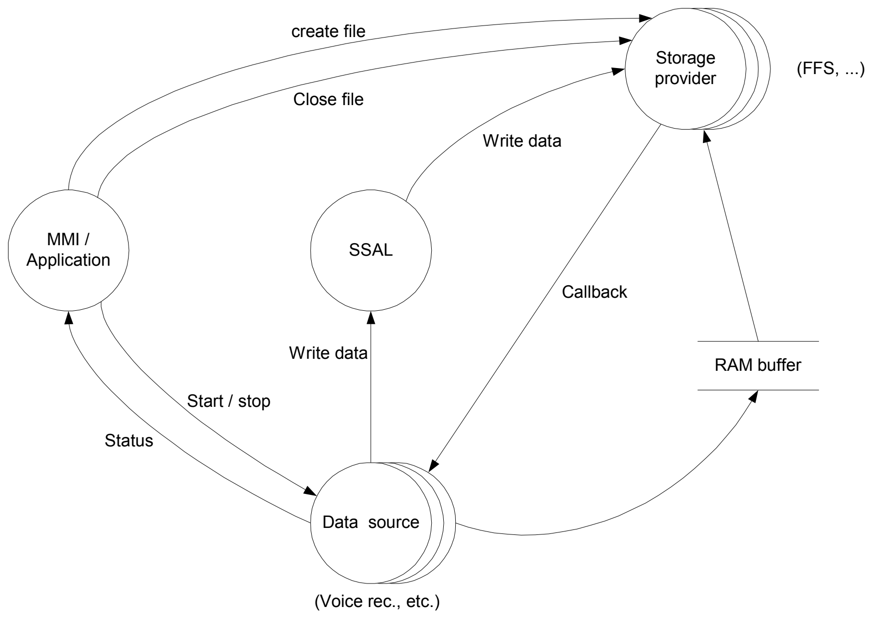


Figure 2.2 Concept for streaming data to FFS

Synchronous option:

The streaming interface can also be used synchronously without callback functions. This mode can be used as an alternative to the normal synchronous interface when high write performance is needed and the file size is unknown at creation time.

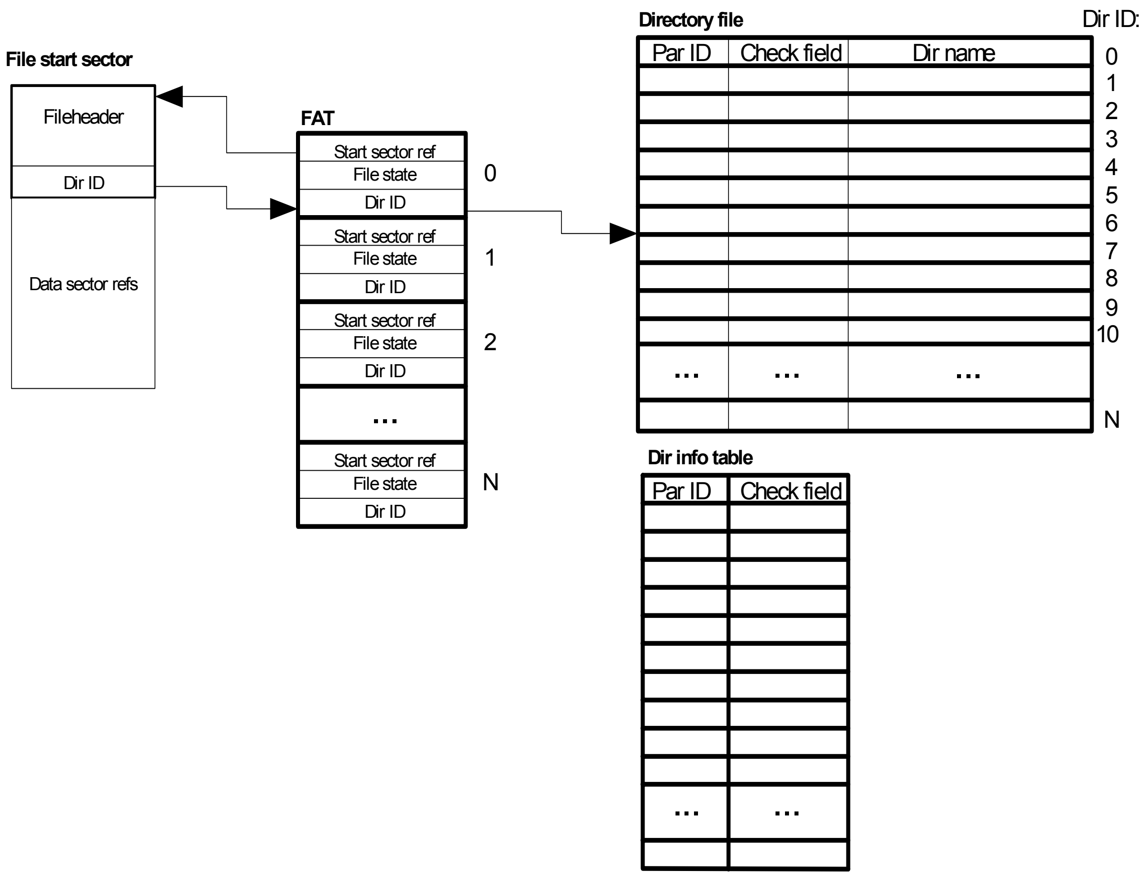
2.6.1 FFS Streaming Buffers

The number of open streaming files is limited because for every open streaming file is related a streaming buffer. Each streaming buffer is of size as a sector size. FFS collects data in the buffer before write and thereby optimizing the write. This buffer operation is necessary for the segmented region flash type.

2.7 Directory support

2.7.1 Directory concept

A dedicated directory file is used to keep track of the directories currently present in FFS, and the directory hierarchy. A Dir Info Table in RAM is used to optimize the searching speed.



Each entry in the directory file uniquely specifies a directory in the FFS. An entry is composed of a dir name, a checksum field, and a parent id. Dir_id for a directory is the index of the directory entry in the directory file.

Dir name: contains only single directory name. Default value: 0xFF (unused)

Check field: Is used to make a fast evaluation when searching for a specific dir. Dir name comparison is unnecessary if the check field doesn't match the check value for the dir being searched. The check field is a simple unsigned 16-bit checksum of the dir name string.

Par ID: Contains the Dir ID for the parent directory. Default value: 0xFFFF (unused). 0x0000: Root directory.

Dir Info Table: To enable faster searching times a part of the Dir file is shadowed in RAM. The Dir Info Table holds the parent id and the checksum value for each directory.

2.7.2 Directory path string conventions

Both DOS and UNIX style directory separators ('\' and '/') are supported.

A string with '\' or '/' as the first character, indicates that the path is specified from the root-directory. In all other cases, the path is specified from current working directory.

'Drive letter:\' e.g. A:\ can alternatively be used to indicate the root directory.

'.\' or './' can optionally be used to specify current working directory.
'..' indicates the parent directory.

Maximum path length: 128 characters (127+\0).

Maximum directory name length: The same as the maximum path length.

Maximum number of files in a folder: The only limit is the maximum number of files in FFS.

Directory hierarchy depth: The depth is limited by the maximum path length (i.e. short directory names would allow a larger depth than long names.

2.7.3 Dir handler

The Dir Handler is an individual internal block, which handles the Dir File and the Dir Info Table. Internal non-reentrant API used by the Pseudo Tasks.

The Dir handler shall take care of the following:

- Create the Dir File during initialization if it doesn't exist.
- Add a new directory to the Dir File and Dir Info Table.
- Remove a directory from the Dir File and Dir Info Table.
- Locate a directory (Dir ID) from a specified path string.
- Rename/move a directory in the Dir File and Dir Info Table.
- Compose and return a complete directory path string given a specific Dir ID
- Support nested searching: findfirst/findnext

2.7.3.1 Dir Handler API

```
ffs_error_code_type dh_init_handler(void)

ffs_error_code_type dh_add_new_dir(ffs_user_type user_id,
                                   const char *path)

ffs_error_code_type dh_remove_dir(ffs_user_type user_id,
                                  const char *path)

ffs_error_code_type dh_locate_dir(ffs_user_type user_id,
                  const char * path,
                  dir_id_type *dir_id,  //Returns Dir Id for a dir or a filename path
                  char *filename,       //If the path is a filename, the name is returned
                  word *file_id,        //Returns File Id if a file was found
                  BOOL *is_dir)         //TRUE if a dir was found

ffs_error_code_type dh_rename_dir(ffs_user_type user_id,
                                  const char *path,          //Specifies the dir to be renamed or
moved                                   const char *new_name)  //New name of dir or path

ffs_error_code_type dh_get_dir_name(dir_id_type dir_id,
                                    const char *path)  //Returns the dir name for the specified Dir
Id
BOOL dh_ffirst_fnext(dir_id_type  start_id,  //Index in Dir File where to start the search
                     dir_id_type  par_id,    //Parent Id of the directory to search for
                     char *name_spec,        //Input search string that may contain wildcards
                     dir_id_type *dir_id,    //Returns the Dir Id if a directory was found
                     char *dirname)          //Returns the dir name if a directory was found
```

2.7.3.2 Dir handler Design

2.7.3.2.1 dh_init_handler

dh_init_handler is called during FFS initialization before the pseudo task scheduler is started, and checks if the Dir-File is present. If not it creates the Dir-File and fills the file with default data by simulating FFS operation requests as described below:

1. Setup the FAT and the pt_mail_recs structure for creating the Dir-File.
2. Set pseudo-task state to the initial create-state.
3. Call the create pseudo-task repeatedly until the create operation finishes or an error occurs.
4. Setup the pt_mail_recs structure for closing the Dir-File.
5. Set pseudo-task state to the initial close-state.
6. Call the close pseudo-task repeatedly until the close operation finishes or an error occurs.
7. Setup the pt_mail_recs structure for writing the Dir-File.
8. Set pseudo-task state to the initial write-state.

9. Call the write pseudo-task repeatedly until the write operation finishes or an error occurs.

2.7.3.2.2 dh_add_new_dir

The dh_add_new_dir function performs the following operations:

1. Check if the specified path complies with the syntax.
2. Search the Dir Info Table and Dir File to check that the directory doesn't already exist.
3. Search the FAT and possible File Headers to check if a file already exists with the specified directory name.
4. Allocate a new Dir Id by searching the Dir Info Table.
5. Initialize the new directory entry in the Dir Info Table.
6. Initialize the new directory entry in the Dir File.
    a. Setup the FAT and the pt_mail_recs structure for writing the Dir-File.
    b. Set pseudo-task state to the initial write-state.
    c. Call the write pseudo-task repeatedly until the write operation finishes or an error occurs.

2.7.3.2.3 dh_remove_dir

The dh_remove_dir function performs the following operations:

1. Check if the specified path complies with the syntax.
2. Search the Dir Info Table and Dir File to check that the directory exists.
3. Check that the directory isn't CWD for any users.
4. Search the Dir Info Table to check that the directory doesn't contain sub-directories.
5. Search the FAT to check that the directory doesn't contain any files.
6. Remove the entry in the Dir Info Table.
7. Remove the entry in the DIR file:
    a. Setup the FAT and the pt_mail_recs structure for modifying the Dir-File entry to the unused values (0xFF).
    b. Set pseudo-task state to the initial modify-state.
    c. Call the modify pseudo-task repeatedly until the modify operation finishes or an error occurs.

2.7.3.2.4 dh_locate_dir

The dh_locate_dir function performs the following operations:

1. Check if the specified path complies with the syntax.
2. Search the Dir Info Table and the Dir File to locate the specified directory.
3. If a dir is found, the dir_id is returned and the is_dir parameter is set to TRUE.
4. If a file is found, the file_id and parent dir_id are returned.

2.7.3.2.5 dh_rename_dir

The dh_rename_dir function performs the following operations:

1. Check if both of the specified paths are complying with the syntax.
2. Search the Dir Info Table and Dir File to check that the original directory exists.
3. Check that the new path (parent path + new name) doesn't exceed max path length
4. Search the Dir Info Table and Dir File to check that the new directory doesn't already exist
5. Search the FAT and possible File Headers to check that a file with the new directory name doesn't exist.
6. Search the FAT to check that the original directory doesn't contain any open files.
7. Check if the dir should be moved or renamed:
   If the directory should be moved:
   - Update Par Id in Dir File and Dir Info Table.
   If the directory should be renamed:
   - Update checksum field in Dir Info Table and dir name and checksum field in Dir File
8. Modify Dir Info Table and Dir File
   a. Setup the FAT and the pt_mail_recs structure for modifying the Dir-File entry.
   b. Set pseudo-task state to the initial modify-state.
   c. Call the modify pseudo-task repeatedly until the modify operation finishes or an error occurs.

2.7.3.2.6 dh_get_dir_name

The dh_get_dir_name function performs the following operations:

1. Read the dir name in the Dir File.
   a. Setup the FAT and the pt_mail_recs structure for reading the Dir-File entry.
   b. Set pseudo-task state to the initial read-state.
   c. Call the read pseudo-task repeatedly until the read operation finishes or an error occurs.
2. Repeat reading dir names in the Dir File until par_id specifies root.
3. Compose a directory path string from the retrieved dir names.
4. If the Dir Id specifies a valid directory, the full path is returned.

2.7.3.2.7 dh_ffirst_fnext

The dh_ffirst_fnext function performs the following operations:

1. Read Parent Id at start index in the Dir Info Table
2. If Parent Id matches the search criteria:
   a. Read the Dir Info Table and Dir File
      i. Setup the FAT and the pt_mail_recs structure for reading the Dir-File entry
      ii. Set pseudo-task state to the initial read-state
      iii. Call the read pseudo-task repeatedly until the read operation finishes or an error occurs
   b. Return dir name and corresponding Dir Id
3. Else: Update start id
4. Repeat step 1-3 until a dir is found or end of file is reached.

2.8 Static and dynamic flash parts.

FFS can be divided into a static and a dynamic part.

Files in the dynamic part can be created, deleted and modified by any user while the MS is running.

Files in the static part are protected against being modified by a user when the MS is running.
This makes the static part useful to store files, which shall be read only. The static part can only be changed by external tools. No FFS functions, which write to the flash, can operate in the static area.
This also means it isn't possibly to create static files dynamically.

2.8.1 Factory default files

The static files can be created as a factory default file. This means they are linked to a dynamic file.

On power up FFS will make a duplicate of the static file if the linked dynamic file doesn't exist.
If the dynamic file is deleted or renamed then FFS will restore the file from the static file.

It is also possibly to request a restoration of one or all files.
It is always possible to restore a factory file by using the id/name of the static file, but if the dynamic file exists then this id/name can be used to restore the file.

If the factory default file is created, as an id file then the linked dynamic file will also be an id file.
If the factory default file is created with a filename then the static file will get '.factory' added to the name. This is handled in the interface functions and will not reduce the maximum length of the filename.

The static factory default files can be read like any other file and the dynamic file can be read/modified like any other dynamic file.

2.9 FFS Memory management.

2.9.1 File type memory pools.

Each file type (except FFS_GENERIC_FILETYPE) in FFS has a dedicated memory-pool to allocate file memory from. The individual pools can only be used for files of the corresponding type.
Therefore, a minimum of memory can be reserved for each file type.
Besides this, there is a global pool common to all file types. If the type uses more than reserved to the type, then it will use memory from the global pool.
The FFS_GENERIC_FILETYPE type will always allocate from the global pool.

The different flash pools will not be separated physically in the flash. The pools and the free space in the pools will be handled logically, by keeping track of the current amount of memory used by each file type.

Since the files in the static part can't be created/deleted/changed runtime, then they don't need to allocate new memory dynamically and they are not counted as a part of the file type pools.

This means if there is reserved 10000 Bytes to a file type, then there is reserved 10000 Bytes to create/change dynamic files of that type.

2.9.2 Physical Memory Pools

FFS keeps track of the physical memory usage by dividing the memory into 3 types.

- Free.
- Reserved, but not yet allocated.
- Used, this includes both valid and invalid (garbage) data.

When a FFS operation needs to allocate new memory it is first moved from the FREE pool to the RESERVED pool. During the operation, FFS will allocate new physical memory from the flash and the memory is moved from the RESERVED pool to the USED pool.
If a FFS operation results in some memory isn't used anymore it will be marked as garbage and when the garbage collector cleans up the memory it is moved from the USED pool to the FREE pool.

2.9.3 Reserving memory

File operations like create, rename, append i.e. needs to allocate new memory. This is a two step check to see if there is enough free space.

1. Check if there is enough free space in the file type pool. This includes the free space in the global pool
2. Check if there is enough free space in the physical free pool

These two checks can cause three different results.

1. There isn't enough free space in the file type pools and the FFS_REQUESTED_MEM_NOT_AVAILABLE error code is returned.
2. There is enough free in the file type pools, but not in the physical pool then the FFS_REQUESTED_MEM_TEMP_NOT_AVAILABLE error code is returned.
3. There is enough free in both pools and the memory is reserved.

If the FFS_REQUESTED_MEM_TEMP_NOT_AVAILABLE is returned, it means there is free space, but too much of it is bound as garbage. Giving the garbage collector time to clean up will free enough space to fulfill the request.
This error code forces the garbage collector to disregard the normal clean criteria's and clean all blocks, which includes garbage.
Although a user makes the task wait for the garbage collector then the user can receive a FFS_REQUESTED_MEM_NOT_AVAILABLE when the garbage collector have cleaned up the memory. This can happen if another task request and reserve memory before the waiting task retries, leaving too little free memory to the waiting task.

2.9.4 FFS Memory allocation strategy

Since the flash blocks have a limited amount of program/erase cycles (100.000 typical), it's necessary to spread the use of blocks equally between all the FFS blocks. To ensure that, a BEC (Block Erase Counter) field is added to each block header to keep track of the numbers of erase cycles for the individual blocks.

When a new block is wanted, the free block with the lowest BEC is always chosen. This strategy does not include the blocks where static files are allocated, but only the dynamic blocks.

When all blocks are written to, except the GC block, then the next block that can be taken as a GC block is only the block that has been just freed by GC operations.

2.10 FFS Management Tables.

2.10.1 FFS blocks.

The flash is physically divided into a number of blocks. FFS_blocks is an array listing the start address of all the flash blocks used to store FFS data.

2.10.2 File Allocation Table (FAT).

A RAM based FAT is used at run-time to hold information about the files stored in the FFS.
In the FAT there is a record per file in the FFS.

A FAT record consist of the following elements:

State: Contains the current state of the file. Possible file states:

- Not used
- Assigned
- Closed
- Open (read)
- Open (read streaming)
- Open (write)
- Open (write streaming)
- Open (write static (only test))
- Open (append)
- Open (modify)
- Open (DRM attributes)
- Create active
- Rename active
- Delete active
- Truncate active

Start sector: Contains the number of the sector containing the file header.

File type: The type of the corresponding file. Makes it faster to use type related search-algorithms.

Interrupts must be disabled when modifying the FAT

The maximum number of files in the FFS must be defined at compile-time, and will define the amount of RAM used for the FAT.

Dir Id: Specifies in which directory the file is located.

2.10.3 Memory Use Table (MUT).

This table contains information about the amount of data currently being used in total and by the different filetypes.

The table shall be updated whenever new data is written to FFS, or when clean up is performed by the garbage collector.

Interrupts must be disabled when modifying the MUT.

The maximum number of file types in the FFS must be defined at compile-time, and will define the amount of RAM used for the MUT.

2.10.4 HW block no Table.

All dynamic blocks in use by FFS are assigned a logical block number. This makes it possible to make wear leveling and a fast garbage collection since the data doesn't have to be copied back to the original address.

The mapping method is called random mapping. Random mapping writes new data onto another (erased) physical block using virtual address mapping technique. The algorithm operates as follows:

a) Write the data onto an erased part of the dynamic FFS flash memory.
b) Modify the logical-to-physical mapping table, the HW block no table, if a new block is used.
c) Mark the old location as invalid.

There is no fixed relationship between the logical block number and the physical block number.

All files uses sector numbers as reference to other parts of the file. This sector number can be used to calculate the logical number of the blocks, which contains this sector. With the HW_block_no table, this logical block number can be converted to physical block no. Otherwise, a logic block number of a block in use can always be read in the block header. The logic block number is written to the block header when the physical block is allocated.

The table shall be updated whenever a new block is taken into use by FFS or when clean up is performed by the garbage collector. The logic block number 0 is reserved for the GC block.

Interrupts must be disabled when modifying the HW_block_no table.

The number of blocks used by FFS must be defined at compile-time, and will define the amount of RAM used for the HW_block_no table.

2.10.5 Sector Usage Table (SUT).

This table contains information about the amount of data currently being used in each block, which sector is the first free sector in the block and how many sectors that contain invalid data.

The table shall be updated whenever a new sector is allocated from a block, a sector is marked as invalid, or when clean up is performed by the garbage collector.

The sector usage table is indexed in logic block numbers for all the dynamic blocks.

Interrupts must be disabled when modifying the sector usage table.

The number of blocks used by FFS must be defined at compile-time, and will define the amount of RAM used for the sector usage table.

2.10.6 Protection of the RAM based tables.

The FFS is very vulnerable against corruption of the FAT, MUT, sector_usage and HW_block_no RAM tables.
If one of these tables is corrupted by unintended writes to the RAM areas, it could result in corruptions of the physical files in the flash or unintended behavior of the FFS in general. To prevent that, the following security precautions are made:

- 2 instances of each of the tables are implemented.
- The 2 instances will be placed in different areas of the RAM.
- When modifying a table it's done in both instances.
- Before activating a flash modifying operation in the FFS task, the 2 instances of each table are compared. If any mismatches are found, all FFS operations will be disabled, and the error code FFS_MEM_TABLES_INCONSISTENCY will be returned. The FFS system is now disabled until the phone has been powered off and on again.

2.10.7 Pre-assigned Id Table (PIT).

It might be that the file-structures or the number of files used for some FFS users are known already at compile-time.

The PIT makes it possible to reserve some id's for these files at compile-time.

When creating new files of this kind, the id is taken from the PIT instead of using *FFS_get_next_free_id().*

The PIT can also be used to store id's for files that are not created during run-time (files that are downloaded to the FFS flash area during production).

It is not possibly to reserve some filenames at compile-time.

2.10.8 User Id Table.

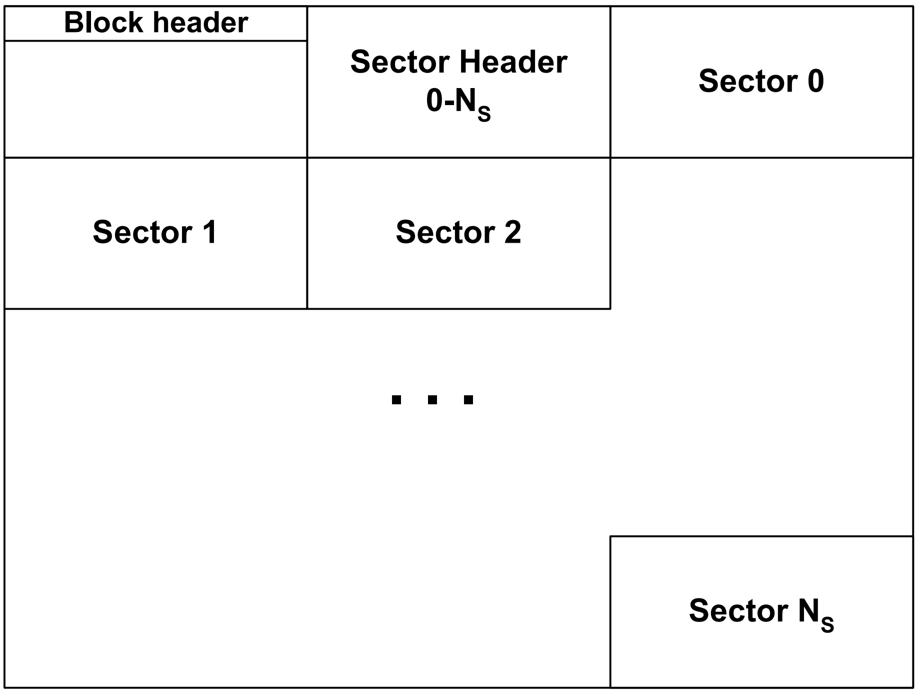
This table contains a unique Id for each process using the FFS. The Id holds the FFS priorities for the different users.

The User Id is used whenever an FFS operation is requested through the function interface, and the FFS task uses the Id to prioritize the requested operations.

2.11 FFS structure in flash memory

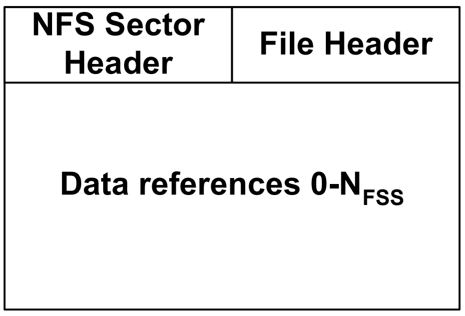
The FFS is using N_B flash blocks (e.g. 64 or 128 Kbytes each).
Each block is divided into a block header containing block status information; a sector header area containing all the sector headers in sequential order aligned with the sectors in the block, and N_S sectors.
Each sector can be a File Start Sector, an Interface Sector or a Data Sector. A Data Sector consists of data only.

The block structure is illustrated below:

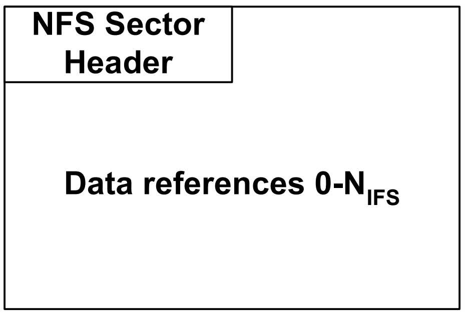


The Block Header is situated at the first bytes of the block. The rest of the first sector in the block is not in use. The second sector (and the numbers of sectors necessary) consists of the Sector Headers. The Sector Headers are always situated in a whole number of blocks. The memory space not used for sector headers in these sectors are not in use.

The File Start Sector consists of a NFS Sector Header, a File Header and data references.



The Interface Sector consists of a NFS Sector Header and data references.

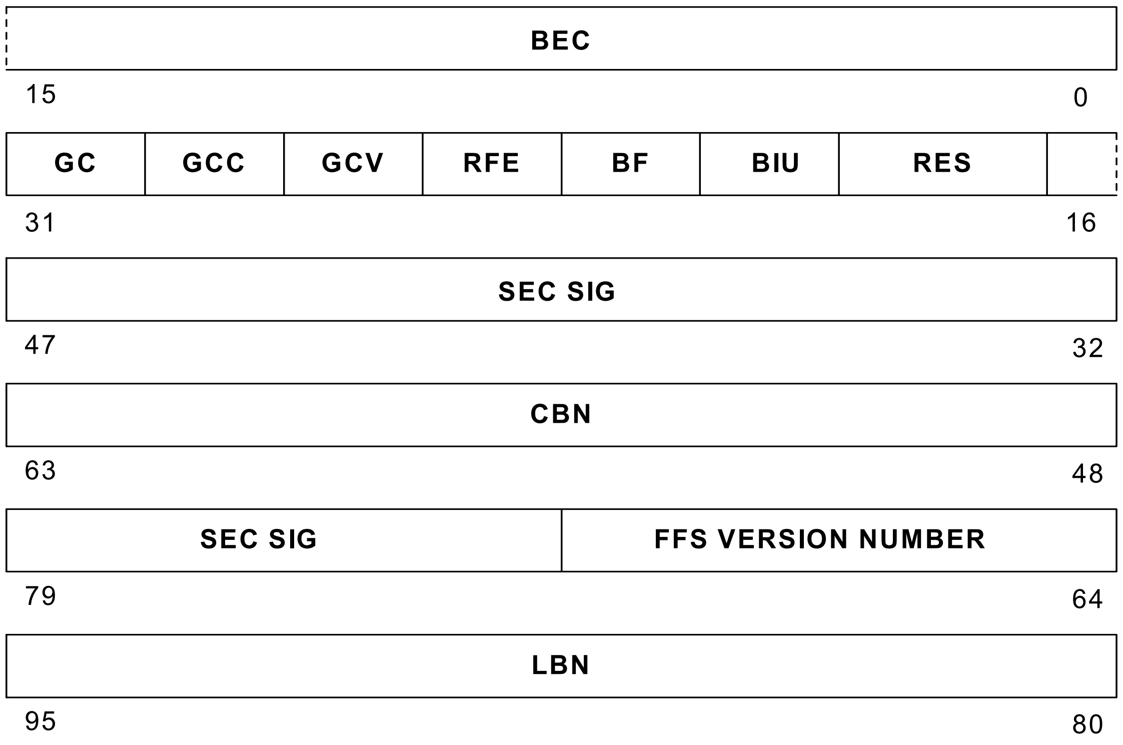


A File Start Sector or an Interface Sector is situated as a sector like the Data Sectors.

What sector header that belongs to what sector is given implicit by the placement of the sector (and the number of sectors). The Sector Headers are placed in the same sequential order as the placement of the sectors. Then the Sector Header 0 refers to the sector 0 and Sector Header 253 refers to sector 253.

2.11.1 Block header.

The block header consists of 96 bits. The format is illustrated below. Each status bits are doubled for security purposes to support multi level flash bit handles.



BEC (bits 0-16)**:** Block Erase Counter.
Keeps track of the number of erase cycles performed on the block.

BIU (bits 20-21)**:** Block In Use.
Is cleared if any sectors are currently programmed.

BF (bits 22-23)**:** Block full.

Is cleared when all sectors in the block are used.

RFE (bits 24-25)**:** Ready For Erase.
When all programmed sectors in the block is marked as invalid the RFE bit is cleared as indication to the garbage collector.

GCV (bits 26-27)**:** Garbage Collector Validated.
All valid data from the block being cleaned have been copied to the garbage collector block.

GCC (bits 28-29)**:** Garbage Collection Completed.
Garbage collection is finished and the block is no longer a garbage collection block.

GC (bits 30-31)**:** Garbage Collector.
Cleared if the block is used as garbage collector block.

CBN (bits 48-63): Clean up Block Number.
Only used by the Garbage Collector block. The number of the physical block currently being cleaned is stored here. 0xFF = None.

SEC SIG (bits 32-47, bits 64-71): Security signature.
Used to guarantee that the content of the block is valid, and not the result of an interrupted block erase operation.

FFS VERSION NUMBER (bits 72-79): FFS version number.
Used to guarantee that the content of the block is valid, and not the result of an interrupted block erase operation. Also used to guarantee that the FFS image is valid.

LBN (bits 80-95): Logic Block Number.
Used as a logic number to make a "random" index of the blocks.

RES (bits 17-19): Reserved for future use.

2.11.2 Sector header.

The sector header consists of 16.
The format is illustrated below. Each status bits are doubled for security purposes to support multi level flash bit handles.

RES	IF	FSS	DNV	DV	SIU
15					0

SIU (bits 0-1)**:** Sector In Use.
Is cleared prior to the programming of the sector.

DV (bits 2-3)**:** Data Valid.
The data in the sector is valid file data.

DNV (bits 4-5)**:** Data Not Valid.
The data in the sector is not valid file data. The file has been deleted.

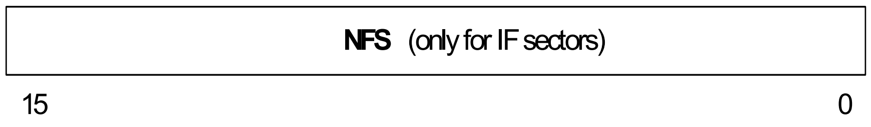
FSS (bits 6-7)**:** File Start Sector.
This bit is cleared if this sector is the start sector for the current file.

IF (bits 8-9)**:** Index File Sector.
This bit is cleared if this sector is an IF sector used to index the data files for the current file.

RES (bits 10-15)**:** Reserved for future use.

2.11.3 NFS Sector header.

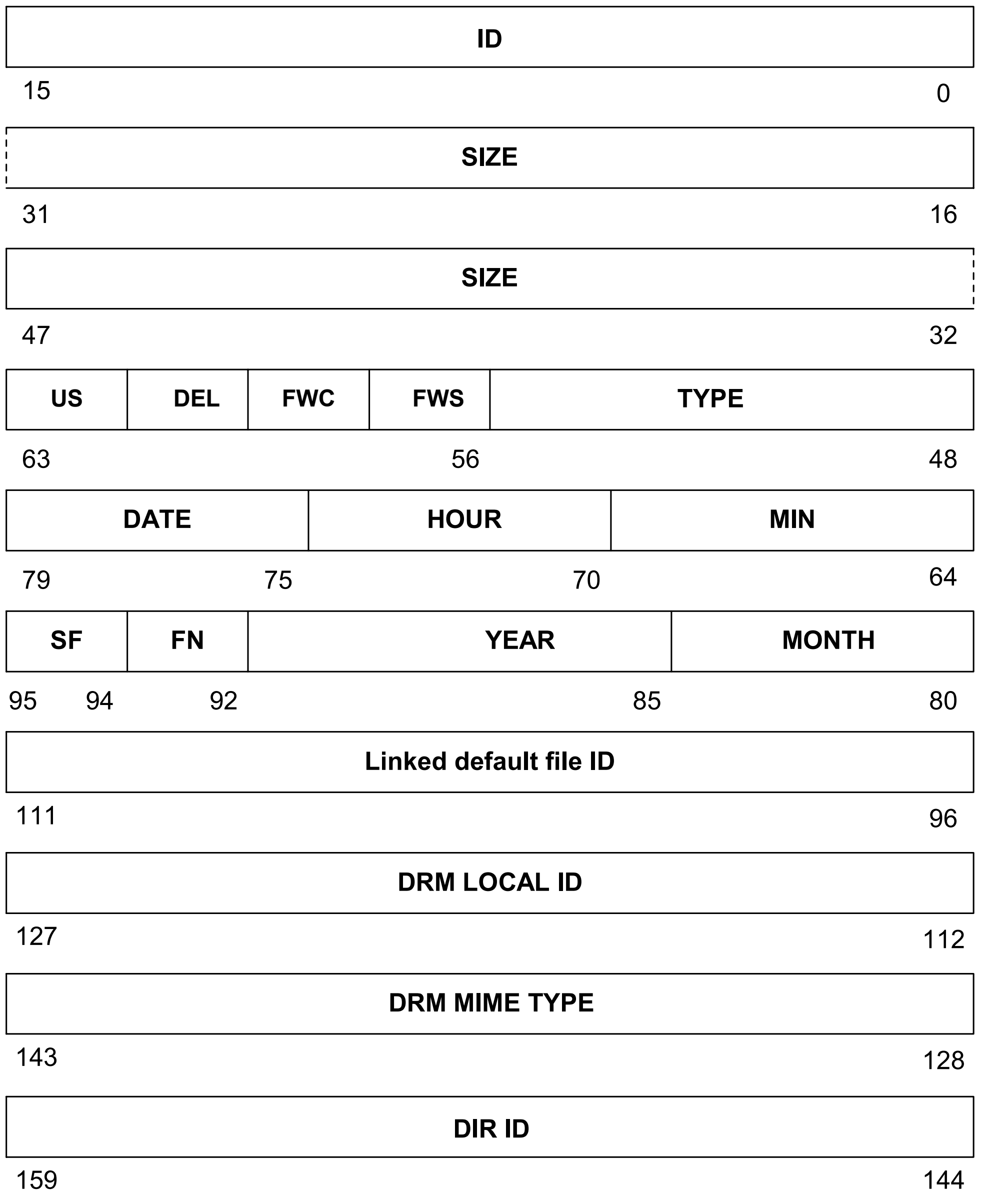
The Index File Sectors (incl. the File Start Sector) consists of 16 bits.



NFS (bits 0-15)**:** Next File Sector.
Contains the logical number of the sector where the file is continued. 0xFFFF indicates EOF.

2.11.4 File header.

The file header is located as the first data in a sector with the FSS bit cleared in the Sector Header, and contains information about the file as shown below. Each status bits are doubled for security purposes to support multi level flash bit handles. The File Header is placed after the NFS Sector Header bits at the File Start Sector and it is long enough to get affected by the segment region design. Therefore two segments gets involved reading the File Header.



ID (Segment 0, bits 0-15)**:** File Id.
Unique integer value.

SIZE (Segment 0, bits 16-47)**:** File size in bytes (max. = 1 Mbytes).

TYPE (Segment 0, bits 48-55)**:** File Type.

FWS (Segment 0, bits 56-57)**:** File Write Started.
The programming of the file is in progress (or finished).

FWC (Segment 0, bits 58-59)**:** File Write Completed.
The programming of the file is completed.

DEL (Segment 0, bits 60-61)**:** Deleted.
The file has been marked deleted.

US (Segment 0, bits 62-63)**:** Update State.
If the file header belongs to an obsolete version of the file, it will be indicated by the US field.
0: File is updated by the append, modify, rename, DRM attribute or truncate operation.
1: File is not updated (valid file header).

MIN (Segment 0, bits 64-69)**:** Timestamp minutes.

HOUR (Segment 0, bits 70-74)**:** Timestamp hours.

DATE (Segment 0, bits 75-79)**:** Timestamp date.

MONTH (Segment 0, bits 80-84)**:** Timestamp month.

YEAR (Segment 0, bits 85-91)**:** Timestamp year since year 2000.

FN (Segment 0, bits 92-93)**:** File Name
The file has a name.

SF (Segment 0, bits 94-95)**:** Static or Dynamic file. If the bit is cleared the file is a static file.

Linked default file id (Segment 0, bits 96-111): If this Id is different from 0xFFFF then the file is a part of a factory default file link.

DRM local id (Segment 1, bits 112-127): DRM local ID attribute.

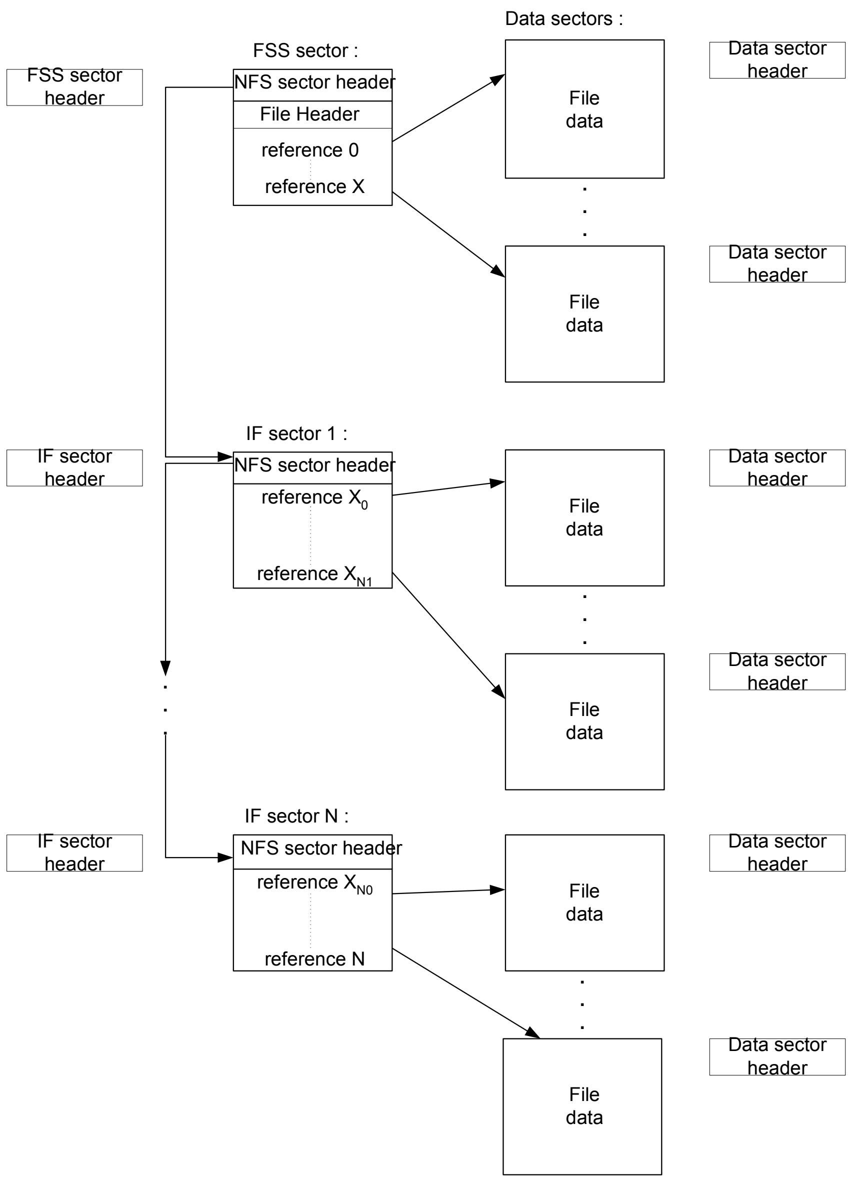
DRM mime type (Segment 1, bits 128-143): DRM mime type attribute.

DIR ID (Segment 1, bits 144-159): Directory ID specifying the location of the file.

2.11.5 Physical file structure.

In the flash, a file is composed by a File Start Sector (FSS), a number of Index File Sectors (IF) and a number of data sectors.
The FSS and IF sectors are linked in a linked list, that contains “pointers” to the data sectors. The “pointers” are absolute sector numbers indexed in a linear relation to the logic block numbers.
The advantage by using the IF sectors, is that all unchanged data sectors can be reused when using the modify- and append operations, then only the corresponding IF sectors and the FSS sector needs to be modified.
For FN (filename) files, the filename is stored in the beginning of the first data-sector.

The physical file-structure is illustrated below:



2.11.6 Memory overhead.

The control headers will naturally give some memory overhead. The size of this overhead depends on the sector size and the size of the files in the system, but can be illustrated in the examples below:

Formulas: The NofDataSectors and NofIfSectors must be rounded up to nearest integer value.

$$NofDataSectors = \frac{FileSize}{SectorSize}$$

$$NofIfSectors = \frac{2 * NofDataSectors + FileHeaderSize}{IfSectorsize - NFSsize}$$

$$OverHead = \frac{(SectorSize + 2 * SectorHeaderSize) * (NofDataSectors + NofIfSectors)}{FileSize} * 100\% - 100$$

File Size	Sector Size	NoDataSectors	NoIfSectors	Overhead
256	32	8	3	54,69%
1024	32	32	6	33,59%
65536	32	2048	294	28,65%
1048575	32	32768	4683	28,58%
256	64	4	1	32,81%
1024	64	16	2	19,53%
65536	64	1024	69	13,41%
1048575	64	16384	1093	13,34%
256	128	2	1	54,69%
1024	128	8	1	16,02%
65536	128	512	17	6,55%
1048575	128	8192	265	6,46%
256	256	1	1	103,13%
1024	256	4	1	26,95%
65536	256	256	5	3,55%
1048575	256	4096	66	3,20%
256	512	1	1	303,13%
1024	512	2	1	51,17%
65536	512	128	2	2,36%
1048575	512	2048	17	1,62%
256	1024	1	1	703,13%
1024	1024	1	1	100,78%
65536	1024	64	1	1,96%
1048575	1024	1024	5	0,88%
256	2048	1	1	1503,13%
1024	2048	1	1	300,78%
65536	2048	32	1	3,33%
1048575	2048	512	2	0,59%

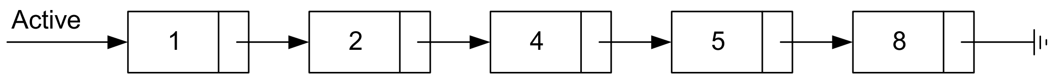
2.12 FFS task.

The FFS task is responsible for handling the bookkeeping, and the updating of the flash contents.

Changes to the FFS are requested through the mail interface where each user process only can have one outstanding mail at a given time.
All mails contain a priority given by the RTOS priority assigned to the requesting process. This means that it shall be possible for the FFS task to check for new incoming mails while an operation is in progress, and if necessary interrupt the ongoing operation while servicing higher priority mails.

2.12.1 Mail handling.

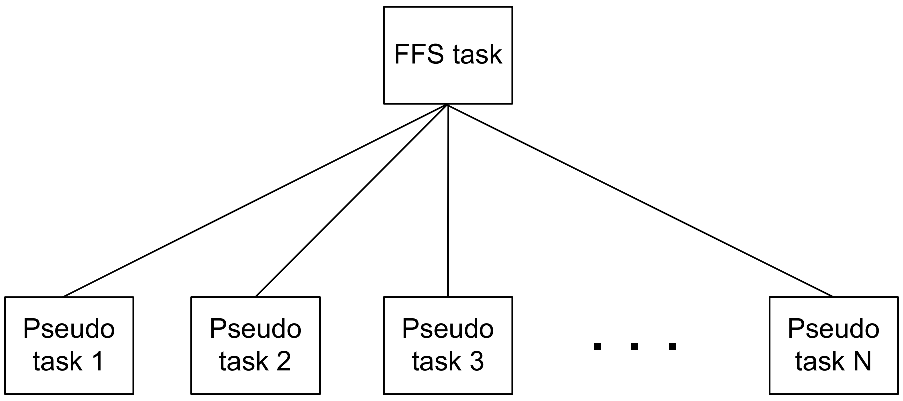
In the FFS task incoming mails will be stored in a prioritized queue, as showed below.



The mail pointed at by *active* is always the mail with the highest priority. New mails are inserted at the position determined by their priority.

2.12.2 Pseudo tasks.

To be able to resume an operation interrupted by an operation with higher priority, pseudo tasks will be used. For each FFS user process there will be a pseudo task in the FFS task.



All pseudo tasks will be executing the same code, but with a separate set of state variables, to ensure that state information is kept during interrupted periods where other pseudo tasks are active.
An array of records is used to hold the state variables (FFS*_state_vars[nof_FFS_users]*).
During execution of the pseudo task code, the mailbox will be checked. If new mails have been delivered the active pseudo task is stopped, and control is given to the mail queue handler that inserts the new mail in the queue, and starts the pseudo task with the highest priority.

2.12.3 Pseudo task operations.

Each pseudo task can activate one of the following operations, determined by the operation type specified in the corresponding mail.

Simplified descriptions in pseudo-code for each of the operations are shown below. The final implementation has to be made so it's possible to interrupt the pseudo tasks and resume them. To detect pending higher priority operations, the mailbox has to be checked on regular basis during execution of the FFS pseudo-tasks.

The following will use the abbreviations: BH = Block Header, SH=Sector Header, FSS= File Start Sector, IFS=Interface Sector and DS= Data Sector.

Common to all pseudo tasks is to find new sectors. They all use the same routine:

```
Find block with free sector
IF (Empty Block) THEN
  Clear the BIU bit in the BH
Clear SIU in SH
IF (Block Full) THEN
  Clear the BF bit in the BH
```

2.12.3.1 Create.

```
IF (there is enough free space) THEN
  Find FSS for the file
  Update FAT, SUT and MUT
  Clear the FSS and IF bits in the FSS SH
  Clear FWS bit in the FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the FSS FH
  IF (filename) THEN
    Clear the FN bit in the FSS FH
    Find first DS for the file and update FSS first data reference
    Write filename
    Clear DV bit in DS SH
  ENDIF
  Clear DV bit in FSS SH
ENDIF
```

2.12.3.2 Write.

If the flash don't have region write restrictions:

```
WHILE (not End Of File) DO
    WHILE (not currently in correct IFS) DO
      Find next IFS
      IF (IFS doesn't exist) THEN
```

```
        Find new IFS
        Update NFS field in the old IFS
        Clear DV and IF bit at IFS SH
      ENDIF
    ENDWHILE
  Read DS reference from IFS
  IF (there is no DS reference) THEN
    Find new DS
    Update reference in IFS
  ENDIF
  Write data to the DS
  Clear DV bit in the DS SH
ENDWHILE
```

2.12.3.3 Write Modify

If the flash has region write restrictions then the write must make a **write modify** when a data sector to write already exists.

```
WHILE (not End Of File) DO
    WHILE (not currently in correct IFS) DO
      Find next IFS
      IF (IFS doesn't exist) THEN
        Find new IFS
        Update NFS field in the old IFS
        Clear DV and IF bit at IFS SH
      ENDIF
    ENDWHILE
    Read DS reference from IFS
    IF (there is no DS reference) THEN
      Find new DS
      Update reference in IFS
    ELSE
      IF (there is enough free space)
        Find new FSS
        Update SUT and MUT
        Clear FSS and IF bits at new FSS SH
        Set US bit at old FSS FH
        Set FWS bit at new FSS FH
        Copy size, type, id, linked id, DRM settings, date and time to the new FSS FH
        IF (filename)
          Clear FN bit at new FSS FH and update the dir id
        ENDIF
        Copy old FSS DS references (except the DS to write to) to new FSS
        WHILE (DS to write to isn't in the FSS/IFS) DO
          Find new IFS
          Clear DV and IF bits at new IFS SH
          Copy old IFS DS references (except the DS to write to) to new IFS
          Update NFS field at the previous FSS/IFS
        ENDWHILE
        Set DV bit in new IFS SH
      ENDIF
    ENDIF
    Write data to the DS
    Clear DV bit in the DS SH
    IF (write modify was done)
      Clear DV bit in new FSS SH
      Clear DNV bit in old DS
      Clear DNV bit in old FSS
      Clear DNV bit in old IFS's
      Update FAT to new FSS
    ENDIF
ENDWHILE
```

2.12.3.4 Write with write buffers

If the flash has region write restrictions then RAM buffers can be used to avoid the write modify when a data sector to write already exists. The data to write is written to a buffer before it is written to the flash. Each of the buffers has the size of a sector/region. If all write buffers are in use and a write is wanted to a data sector that already exists then the write modify must be used.

A write buffer is allocated when in a FFS write a write to a sector is wanted and the size of the data to write would not out fill a whole sector. The data is written to the write buffer. The number of bytes of data written to a write buffer is counted. When the number of data has reached the write buffer size or the file is closed, then the FFS data will be flushed from the write buffer to the actual sector in the flash and the write buffer will be freed. If there has been any overwrite of data in the sector then the write buffer will get flushed even if the sector isn't filled (counting the number of bytes). This is accepted to get a simple handle of the write buffers. If a rewrite is made to the sector after the flush then a write modify operation is necessary.

A detailed description of writing with write buffers is made in the state-event diagram, see Appendix 5.4.

2.12.3.5 Append.

```
IF (there is enough free space) THEN
  Update FAT, SUT and MUT

  Find new FSS
  Clear the FSS and IF bits in the new FSS SH
  Set US in the old FSS FH
  Clear FWS bit in the new FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the new FSS FH
  IF (filename) THEN
    Clear the FN bit in the new FSS FH
  Copy data references from the old FSS to the new one.
  Clear DV bit in the new FSS SH

  Locate the last DS used by the file

  Find new DS
  Update new DS reference in FSS/IFS
  Copy reusable data from the corresponding old DS
  Write new data to the DS
  Clear DV bit in the DS SH

  WHILE (not end of new data) DO
    IF (FSS/IFS is full)
      Find next IFS
      Update NFS field in the previous IFS
    ENDIF
    Find new DS
    Update DS reference in IFS
    Write data to the DS
    Clear DV bit in the DS SH
  ENDWHILE

  Clear the FWC bit in the new FSS FH
  Clear the DV bit in the new FSS SH
  Clear DNV in all not reused DS and IFS SH
  Clear DNV bit in the old FSS SH
ENDIF
```

2.12.3.6 Truncate.

```
IF (there is enough free space) THEN
  Update FAT, SUT and MUT

  Find FSS
  Clear the FSS and IF bits in the new FSS SH
  Set US in the old FSS FH
  Clear FWS bit in the new FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the new FSS FH
  IF (filename) THEN
    Clear the FN bit in the new FSS FH
  Copy reusable DS references from the old FSS to the new one.
  Clear DV bit in the FFS SH

  WHILE (not end of reusable data) DO
    Find next IFS
    IF (IFS doesn’t exist) THEN
      Find new IFS
      Update NFS field in the previous IFS
      Copy reusable DS references from the corresponding old IFS.
    Clear DV bit in the IFS SH
   ENDIF
  ENDWHILE

  If (new file size  != DS boundary) THEN
   Find new DS
   Update DS reference in FSS/IFS
   Copy reusable data from the corresponding old DS
   Clear DV bit in the DS SH
  ENDIF

  Clear the FWC bit in the new FSS FH
  Clear the DV bit in the new FSS SH
  Clear DNV in all not reused DS and IFS SH
  Clear DNV in the old FSS SH
  Update the FAT
ENDIF
```

2.12.3.7 Modify.

```
IF (there is enough free space) THEN
  Update FAT, SUT and MUT

  Find FSS
  Clear the FSS and IF bits in the new FSS SH
  Set US in the old FSS FH
  Clear FWS bit in the new FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the new FSS FH
  IF (filename) THEN
   Clear the FN bit in the new FSS FH
  Copy reusable DS references from the old FSS to the new one.
  Clear DV bit in the new FSS SH

  WHILE (not end of modify data) DO
   WHILE (not currently in correct IFS) DO
    Find next IFS
    IF (IFS doesn't exist) THEN
     Find new IFS
     Update NFS field in the previous FSS/IFS
     Copy reusable DS references from the corresponding old FSS/IFS.
     Clear DV bit in the IFS SH
    ENDIF
   ENDWHILE
   Find new DS
   Update DS reference in FSS/IFS
   Copy reusable data from the corresponding old DS
   Write new data to the DS
   Clear DV bit in the DS SH
  ENDWHILE

  Clear the FWC bit in the new FSS FH
  Clear the DV bit in the new FSS SH
  Clear DNV in all not reused DS and IFS SH
  Clear DNV in the old FSS SH
  Update the FAT
ENDIF
```

2.12.3.8 Close (write mode).

```
WHILE (not End Of File) DO
  Allocate new DS and IFS if they aren't allocated.
  IF (IFS created)
    Clear the IF bit in the IFS SH
    Update the NFS field in the previous FSS/IFS
  ENDIF
  Clear the DV bit in the DS or IFS SH
ENDWHILE
Update the FAT
Clear the FWC and DV bits in the FSS FH
```

2.12.3.9 Delete.

```
Update FAT
Clear the DEL bit in the FSS FH
Clear the DNV bit in all IFS and DS SH's
Clear the DNV bit in the FSS SH
Update FAT, SUT and MUT
```

2.12.3.10Rename.

```
IF (there is enough free space) THEN
  Update MUT and SUT

  Find FSS
  Clear the FSS and IF bits in the new FSS SH
  Set US in the old FSS FH
  Clear FWS bit in the new FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the new FSS FH
  IF (filename) THEN
    Clear the FN bit in the new FSS FH
  Copy DS references from the old FSS to the new one (except the first DS reference if filename)

  IF (filename and the name is changed) THEN
    Find new DS
    Write new filename
    Copy the rest of the old first DS to the new first DS
    Write DS reference in the new FSS
    Clear DV bit in the DS SH
  ENDIF

  Clear the FWC bit in the new FSS FH
  Clear the DV bit in the new FSS SH
  Update FAT, SUT and MUT
  Clear DNV bit in the old FSS SH
ENDIF
```

2.12.3.11 Copy.

This pseudo task will be reusing other pseudo tasks

```
Open old file
Create new file

WHILE (not End Of File) DO
  Read from old file
  Write to new file
ENDWHILE

Close new file
Close old file
```

2.12.3.12 DRM attributes.

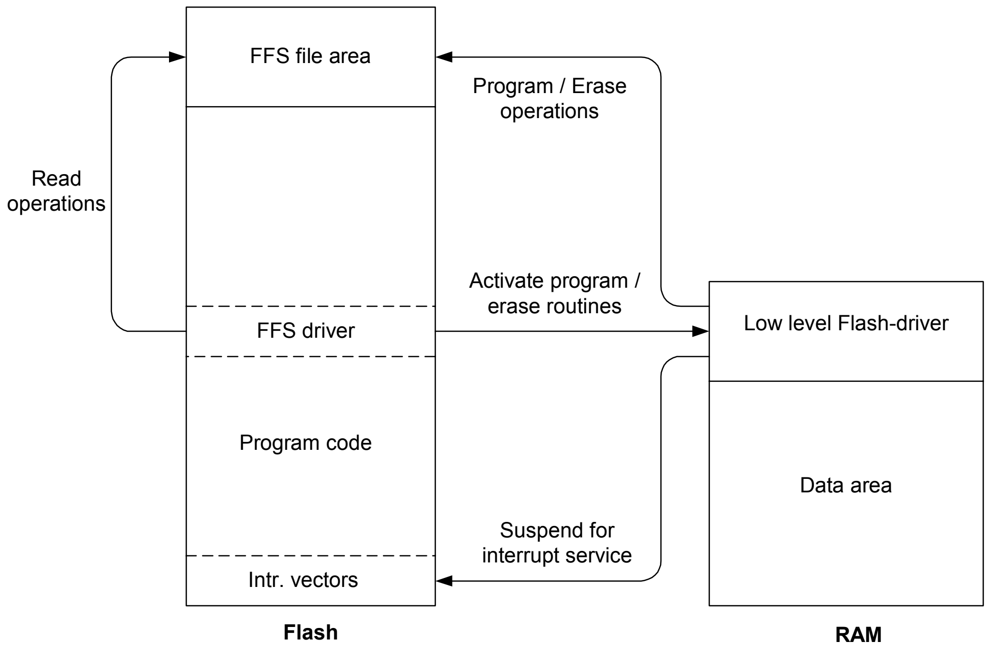
```
IF (there is enough free space) THEN
  Update MUT and SUT

  Find new FSS
  Clear the FSS and IF bits in the new FSS SH
  Set US in the old FSS FH
  Clear FWS bit in the new FSS FH
  Update the fields: Id, size, type, security, linked id and timestamp in the new FSS FH
  IF (filename) THEN
    Clear the FN bit in the new FSS FH
  Copy reusable DS references from the old FSS to the new one.

  Clear the FWC bit in the new FSS FH
  Clear the DV bit in the new FSS SH
  Clear DNV in the old FSS SH
  Update the FAT
ENDIF
```

2.13 Low-level code and interrupt handling.

FFS performs all programs and erases to the flash media through a RAM based low-level flash driver. See illustration below.



The basic functions provided by the low-level flash driver are: program, erase, program/erase suspend, and interrupt polling. During a flash program or erase, interrupts with vectors in flash are disabled and control is turned over to a RAM based low-level flash driver. This routine polls interrupts while monitoring the progress of the flash program or erase operation. If a higher priority interrupt occurs or a timeout of app. 1 msec. is reached, the RAM based function suspends the flash operation and allows the interrupt handler to execute from flash. The timeout is added to allow low priority interrupts, which is not polled, to be executed. Upon completion of the interrupt routine, the flash program/erase operation is resumed by the RAM based flash driver.

The maximum program/erase suspend time for a typical flash is 20 µs.

The low-level driver will be shared with the EEPROM emulation driver.

2.14 Garbage collection.

When a file is deleted or modified in the FFS, the flash memory, which isn't used any more, is marked as invalid ("DNV" bit in the sector header). The data is not instantly removed in the flash, so it has to be removed physically to free the obsolete memory to be used again by new files. The physical clean up is done by the garbage collector running at a low priority task in the system.

The garbage collection can run in parallel with FFS operations with the exceptions when verification takes place or a new GC block is allocated. These exceptions in the garbage collection have been protected by raising the RTOS priority for the GC process to the same priority as the FFS process. Thereby the FFS process is blocked during these critical sections. The GC priority will only be raised for short periods of time, to prevent that FFS operations and processes with priority between FFS and GC are being blocked.

There shall always be a free block allocated for the garbage collector, to avoid dead locks when clean up is needed.

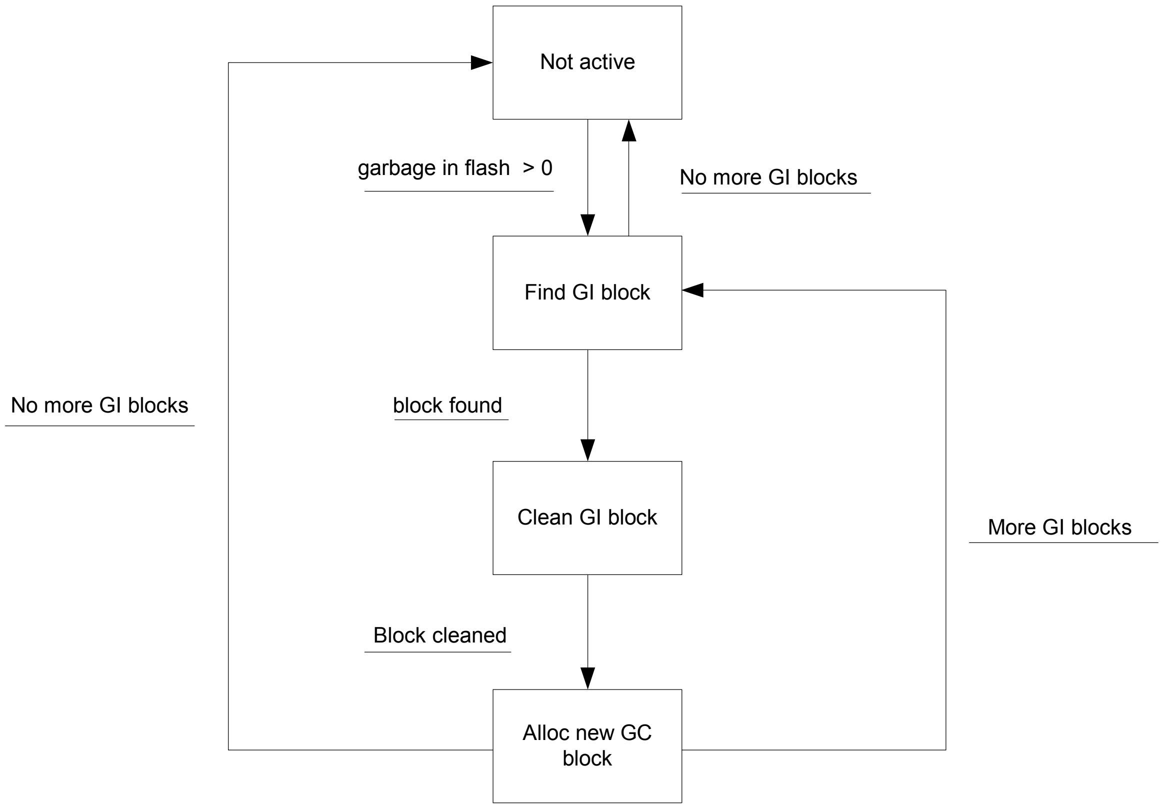
2.14.1 Garbage collection tasks

The tasks for the garbage collector:

1. Detect if there are blocks with invalid sectors. Recognized by the sector usage table that is updated if DNV bit in the sector header is set.
2. Remove garbage using the garbage collector block.
3. Update the block erase counter when erasing blocks.
4. Check if the ram tables are valid.

The garbage collection is described in the state machine below.

For more information about garbage collection operation, the garbage collection state machines can be seen at Appendix 5.2.



2.14.1.1 Not active state.

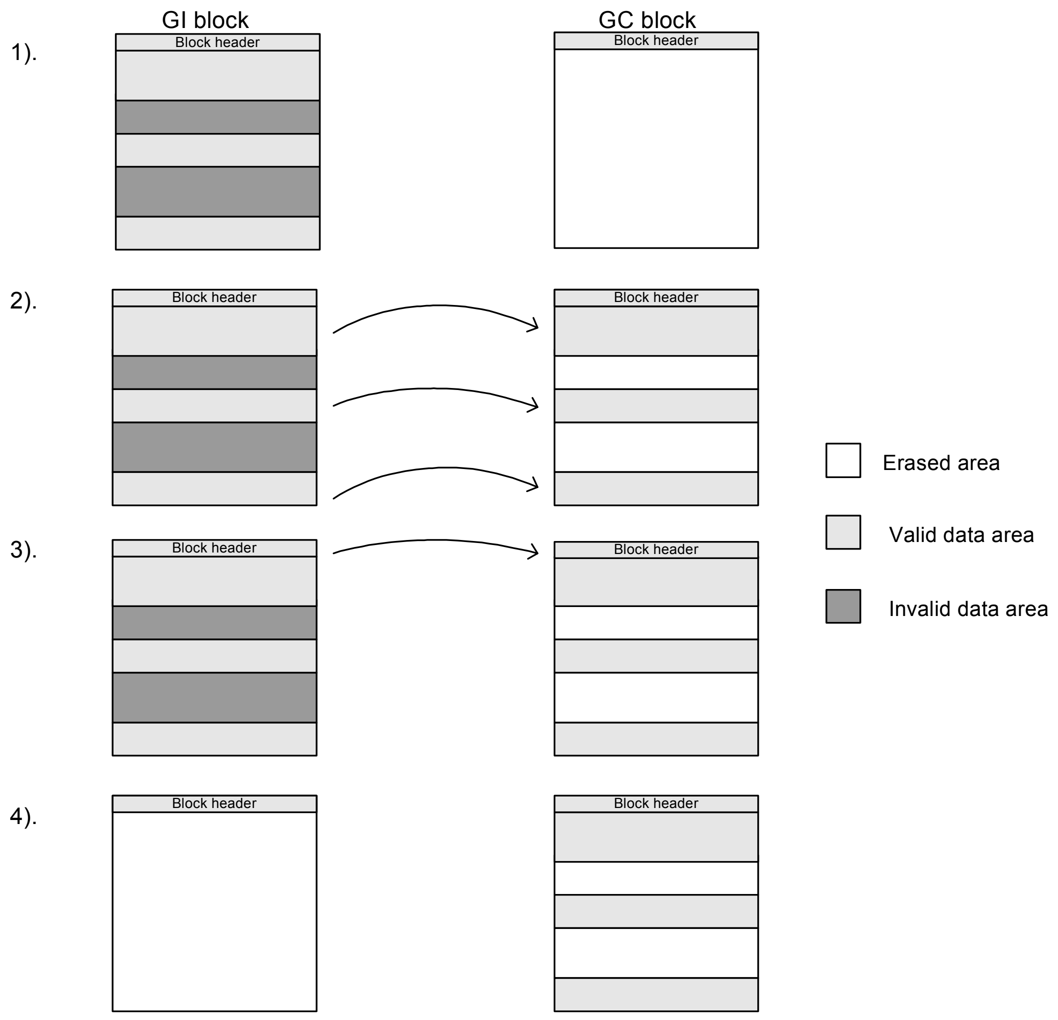
Initial state. When a garbage collection operation is completed, the state machine will return to the not active state. Whenever the garbage collector is activated, it will check if the ram tables are valid.

2.14.1.2 Find GI block state.

Locate a block containing garbage (if any), by checking out the SUT.

2.14.1.3 Clean GI block state.

The block clean up procedure is illustrated below.



The block containing garbage (GI block) is cleaned, by copying the valid sectors to the GC (garbage collector) block. After all valid sectors have been copied, the GC block is marked as the same logical block as the GI block and FFS will now see the GC block as the same block as the GI block instead of the GI block. Then the GI block is erased.

The block header bits are used to keep track of the progress, and to make sure that power loss recovery is possible (if power is lost during clean up).

Clean up procedure (see illustration above)**:**

1. The GC block is a block like any other free block as long as the cleanup hasn't been initiated. The BEC and security signature are updated in the block header. Notice no LBN has been set. The GC always has the logic block number 0 in the hw_block_no ram table but this 0 is not written to the GC block header. The block is only reserved in RAM in the hw_block_no table where it is written for the hw_block_no[0].

When the cleanup is initiated then the GC bit is set in the block header. This must be done to be able to detect this block as a garbage collector block where data has been copied at power loss recovery.

2. All used sectors not marked as invalid sectors in the GI block are copied to the GC block including sector and file headers. Prior to copying the sectors the BIU (Block In Use) bit is cleared in the GC block header to indicate that a clean up operation is in progress.

The GC block header is modified.

a. The LBN (Logical Block Number) field in block header of the GC block is updated with the number of the GI block.
b. The CBN (Cleanup Block Number) field in block header of the GC block is updated with the number of the GI block.

3. It is verified that FFS didn't write in GI while copying the valid data. If FFS did write in GI while GC was copying data, then the copy procedure (2) is repeated.

The block headers are modified.

a. The GCV (Garbage Collection Validated) bit is cleared in the GC block header to indicate that all valid data is now stored in the GC block. The sector_usage, hw_block_no and mut tables are updated accordingly. After this point FFS will see GC as the GI block and stop using the previous GI block.
b. The RFE (Ready For Erase) bit is cleared in the GI block header to mark it is ready to be erased.

4. The GI block is erased and the BEC (Block Erase Counter) field is incremented by 1. The security signature is updated.

The GCC (Garbage Collection Completed) bit is cleared in the GC block indicating that the block is no longer used as garbage collection block.

2.14.1.4 Alloc new GC block state.

A new GC (garbage collector) block is selected between the empty blocks in the system. The one with the lowest number in the BEC (Block Erase Counter) field is selected. The hw_block_no table is updated accordingly.

2.14.2 Forced garbage collection

The purpose of forced GC is to ensure that file-operations, which need to allocate new memory, are not blocked by garbage that has been accumulated in FFS.

The background cleanup is running from low priority (GC process), therefore it is possible that garbage will accumulate in situations with high CPU load and high FFS activity. If the FFS store gets full in this situation, it will not be possible to allocate memory before some garbage has been cleaned.

To overcome these situations, forced GC will be activated from FFS process level. When forced GC is active, the FFS blocks will be cleaned one by one, until the requested amount of memory, needed by the requested operation, has been cleaned. After freeing the memory the ongoing FFS operation

will continue seamless for the FFS-user. Note: the time to perform the clean up, will extend the total time for carrying out the requested operation.
Forced GC is done from a special forced GC pseudo task in the FFS process. This pseudo task has the same user-priority as the original operation request. During forced GC, it will still be possible to schedule file operations requested by higher priority FFS-users.

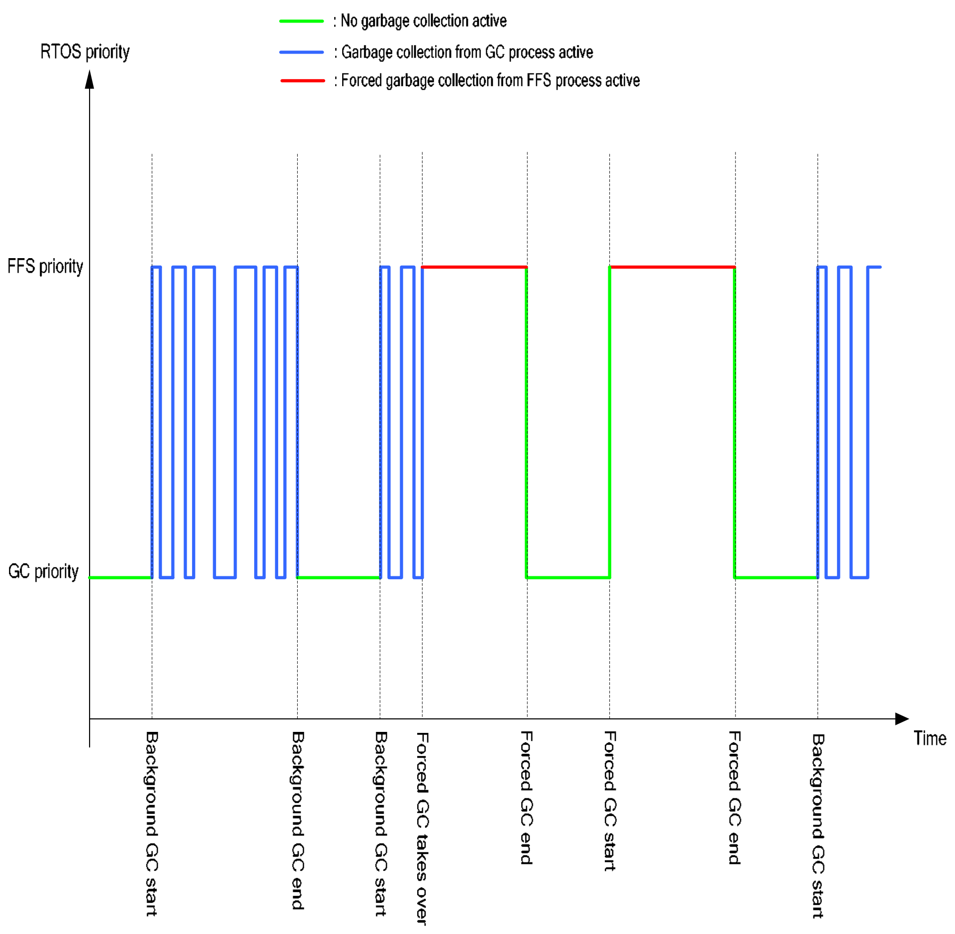
Forced GC from the FFS process can run a full garbage collection sequence, when activated when the background GC is idle, but it also has the capability to take over an already ongoing background garbage collection sequence. Garbage collection is implemented as a state machine, which makes it possible for the forced GC routines to determine the actual state of an ongoing background GC sequence. State-variables also makes it possible, for the background GC process to detect if an ongoing garbage collection operation has been finished by forced GC. In this case, the background GC considers the actual operation to be done and restarts from the beginning with a new GI block to clean if there are any GI blocks found. Otherwise, it switches to the idle state.

During the background garbage collection sequence, there are some critical phases, where the FFS process must not be activated:

- During updates of block headers
- During sector copying form GI to GC
- During sector verification after copy
- When block erase is active (will only be scheduled for small periods of time)
- When updating global memory tables

To protect these critical sections, the background GC process temporarily raises the RTOS priority to the same level as the FFS process. Thereby the FFS process is blocked during these critical sections. The GC priority will only be raised for short periods of time, to prevent that FFS operations and processes with priority between FFS and GC are being blocked.

The figure below shows the sequence of background and forced GC, and the situation where forced GC takes over from the background GC.



In Appendix 5.3 the forced GC state machine are shown.

2.15 Wear Leveling

Allocation of a data segment in a new block is limited to a choice of block between the free blocks. If very static data are saved in the dynamic blocks then these blocks can be occupied a long time. This occurrence can make some blocks keep a low erase count while others approaches the flash limit lifetime very fast. To make these blocks with “dead data” sometimes appear in the free blocks selection pool, wear leveling has been added. With wear leveling, a block that hasn’t been erased as much as other blocks, because it contains data that are changed seldom, are moved to another block that has been erased often. The old block is erased for future use. Wear leveling operations only occurs on dynamic blocks. Static blocks are kept untouched after being programmed.

2.15.1 Decision criteria’s for wear leveling

Wear leveling must run as seldom as possible but still be in operation. If wear leveling is running too often then blocks will be erased unnecessary an thereby only decreasing the flash lifetime. If wear leveling is run too seldom then the flash lifetime is also decreased because “static data” blocks are kept with very low erase counts when other blocks are erased often making these blocks reach the maximum flash lifetime long before others.

The decision is made with the difference between the minimum and the average BEC. It is chosen to use the average not forcing wear leveling operations because of spikes on the max BEC.

To minor the numbers of wear leveling at the beginning of the flash lifetime the wear leveling decision level is initiated as a value that is decreased as the flash lifetime is reached. The decrease is made linear to simplify the implementation and get a smooth decrease.

Another criteria to make wear leveling is if it is possible to find a block that contains so much “static” data that it will be left untouched by the Garbage Collector. To detect a block with “dead” data the following decision criteria has been used:

- The block must be in use
- The BEC for the block must be the lowest between other blocks that can fulfill the wanted criteria’s
- The number of invalid and free sectors must be less than the garbage collector criteria at 10 %.
- It must be possible to find the same block after garbage collection operation a number of times.

When a block has been found possible to make wear leveling at then this block is kept as the wanted block to wear leveling even though other blocks with lower BEC counts are set in use. This is done to be able to count the block as still being with “static” data a number of times.

Wear leveling tests are described in Appendix 5.1

2.15.2 The Wear Leveling operation

With wear leveling, data from a block, detected with “static” data and a low BEC, are moved to a block with a larger BEC. The block where data are moved to is found with the following conditions:

- The block must NOT be in use. The selected garbage collection block is possible to be chosen.
- The BEC for the block must be the largest between other blocks that can fulfill the wanted criteria

To be able to implement the upper criteria’s to find a destination block then all blocks that are not in use must be without any status bit set. The block reserved for the garbage collector is only reserved in RAM. No status bits must be set in the garbage collection block header because it must be possible to choose another block than the actual chosen garbage collection block to clean up the block for wear leveling.

Wear leveling uses the same entries as garbage collection with exception of the initiate start state conditions. The initiate start state is GC_CLEANUP_WL. This Wear leveling start state initiates a new search for a “garbage collection” block that fulfills the wear leveling conditions for a destination block. The previously found garbage collection block number at the ram table hw_block_no is simply overwritten with the newly found wear leveling destination block. Wear leveling continues with the same operations as garbage collection. If there is any invalid data present in the source block for wear leveling then it is "cleaned" as a GI block is "cleaned" at garbage collection.

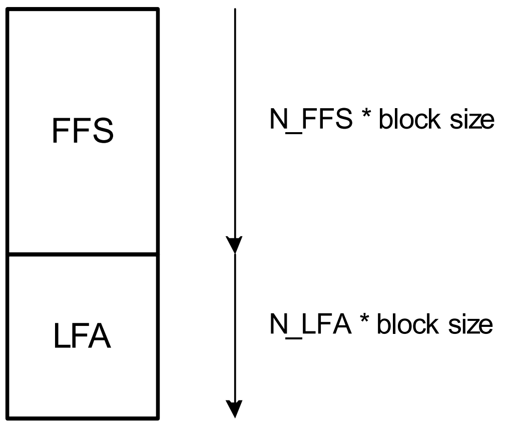
If forced garbage collection is initiated while wear leveling is operating then the wear leveling operation will be continued just like a garbage collection operation will be continued by forced garbage collection.

2.16 Linear File Access (LFA).

Some application might want to access a file using a fast and linear approach, but the size of the file makes it undesirable to load it into RAM memory. This can e.g. be a wallpaper image.
LFA offers a method to store files in flash without the segmentation used by the normal FFS operations.
The files in LFA can be accessed with a pointer just as if it was loaded to RAM.

2.16.1 FFS LFA Memory extension.

To support this feature of linearity the FFS is extended with separate blocks predefined for linear file access.



The linear file access memory is defined as a number of blocks. If a file is copied from the FFS to the LFA the file will occupy one or more blocks (e.g. N * block size). This is done to simplify the process of erase and write to the block when the file is changed to another file copied from the FFS.

2.16.2 FFS LFA Memory Pools.

The blocks of the LFA memory are divided in LFA types like wallpaper and melody. This means that blocks are being predefined to hold a specific type of file, like a wallpaper. A type will only consist 1 file. This is made for simplicity purposes to make it very easy for the user to access the file for reading. The user don't have to know the actual file id or name to read it, only the LFA type is necessary.

2.16.3 FFS LFA.

To make use of the LFA

- The memory for the LFA must be defined.
- The user shall copy a file from the FFS to the LFA memory area with dedicated LFA functions (FFS_load_to_LFA_fn() or FFS_load_to_LFA()). If the LFA memory area isn't empty then the memory area is erased before the write. The file is opened for read operation, then the file is copied from the FFS to the LFA memory area, and then the file is closed.
- The user can at any time access the file reading directly with a pointer address to the file data. The pointer address to the file data is given with a dedicated LFA function (FFS_get_LFA_ref()).If the memory area is empty then a NULL reference pointer is returned.
- The user can ask for info about the file placed at the LFA memory area with a dedicated LFA function (FFS_get_LFA_info()). The file id and file size is copied to the LFA block header with the file. This file id is used to get file info of the file at the FFS.

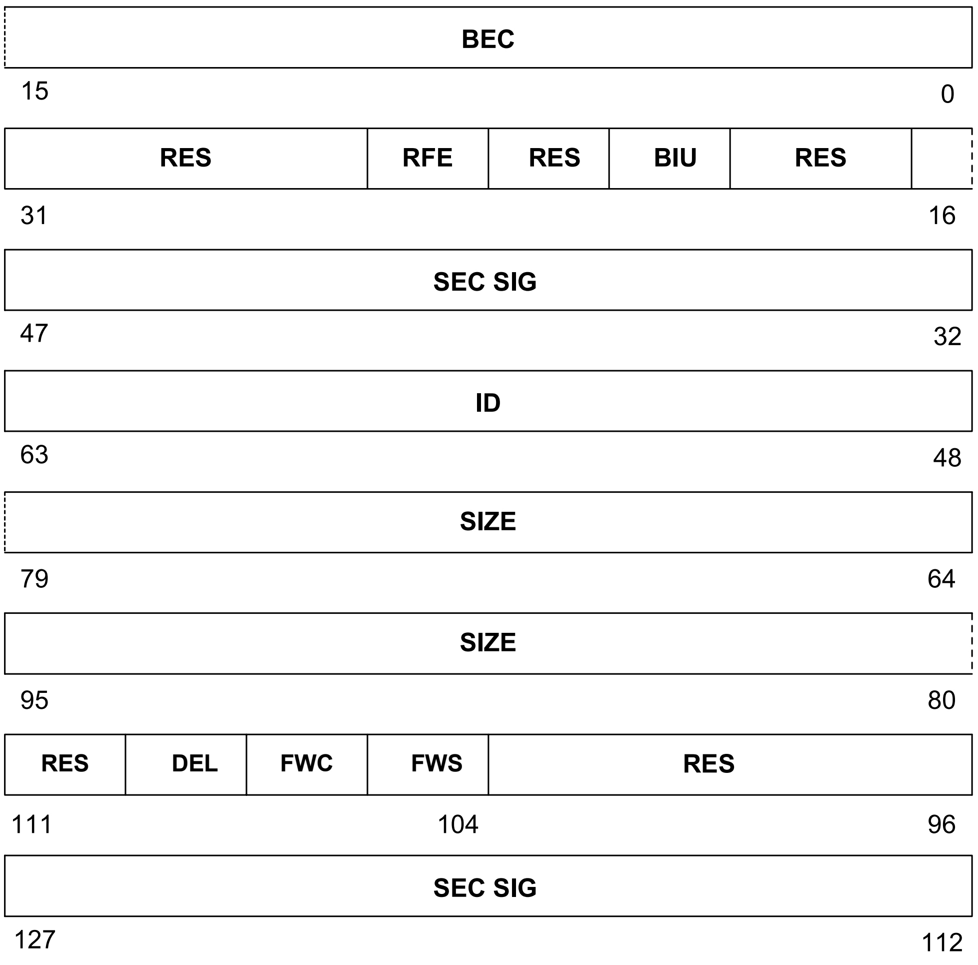
It is not possible to copy the file from the LFA memory to the FFS.

2.16.4 FFS LFA memory Clean.

The LFA memory will be cleaned every time the user uses the FFS_load_to_LFA..() functions. The cleaning of memory at the LFA memory area is done without use of the garbage collector for the FFS. The clean will be done directly from the LFA load pseudo task. To avoid that this clean will stall all other pseudo tasks for the FFS (also higher prioritized) the clean is working in cleaning intervals where the pseudo task can be temporarily released.

2.16.5 LFA block headers.

For all defined LFA types there will be a type block start header. It is a reduced version of the block header from the FFS with some add on of the file header from the FFS. Each status bits are doubled for security purposes to support multi level flash handles.



BEC (bits 0-16)**:** Block Erase Counter.
Keeps track of the number of erase cycles performed on the block.

BIU (bits 20-21)**:** Block In Use.
Is cleared if the LFA block is currently programmed.

RFE (bits 24-25)**:** Erase started (Ready for erase)
Cleared before erase. When the block has been erased then the bit is set high again. The RFE bit is checked at power up. If power have been removed before the erase was completed then the erase must be completed at power up.

ID (bits 48-63)**:** File Id.
Unique integer value.

FWS (bits 104-105)**:** File Write Started.
The programming of the file is in progress (or finished).

FWC (bits 106-107)**:** File Write Completed.
The programming of the file is completed.

DEL (bits 108-109): The file has been deleted from the FFS and is now only present at the LFA block memory.

SIZE (bits 64-95)**:** File size in bytes (max. = 1 Mbytes).

SEC SIG (bits 32-47, 112-127): Security signature.
Used to guarantee that the contents of the block is valid, and not the result of an interrupted block erase operation.

RES (bits 17-19, 22-23, 26-31, 96-103, 110-111): Reserved bits for future use.

If
    SEC SIG is valid
    BIU, FWS and FWC are cleared
    RFE is not cleared
Then a valid file has been copied to the LFA block and the reference can be set to point to data after the LFA block header. Notice that the file time stamp, file name and file type are not copied to the LFA memory. Only the file size and the file id are copied. The file size is used when reading the file to know how many bytes to read from the LFA memory block. The file id makes it possible to get file info about the file by finding the file at the FFS.

2.16.6 Use of LFA at power up.

If a pointer is used to access a file in the LFA memory area then this pointer must be updated at power up.

For the LFA blocks, it is critical to loose power when a file is being written to the LFA memory and when the LFA block is being erased.

Before a file is written to the LFA memory, the BIU and FWS are cleared. After the file has been written, FWC is also cleared. At power up both BIU, FWS and FWC must be cleared. If BIU and FWS are cleared and FWC isn't at power up then a file write has been interrupted by a power loss. Then the LFA block memory must be cleared and reloaded with a new file.

Before an erase of a LFA block, the RFE bit is cleared. If the RFE is cleared at power up then an erase of the LFA memory block have been interrupted by a power loss. Then the LFA block memory must be cleared and reloaded with a new file.

If the SEC SIG is not valid at power up then the LFA block memory is cleared.

2.17 Multi tasking issues modifying flash storage

2.17.1 Multi tasking at erase

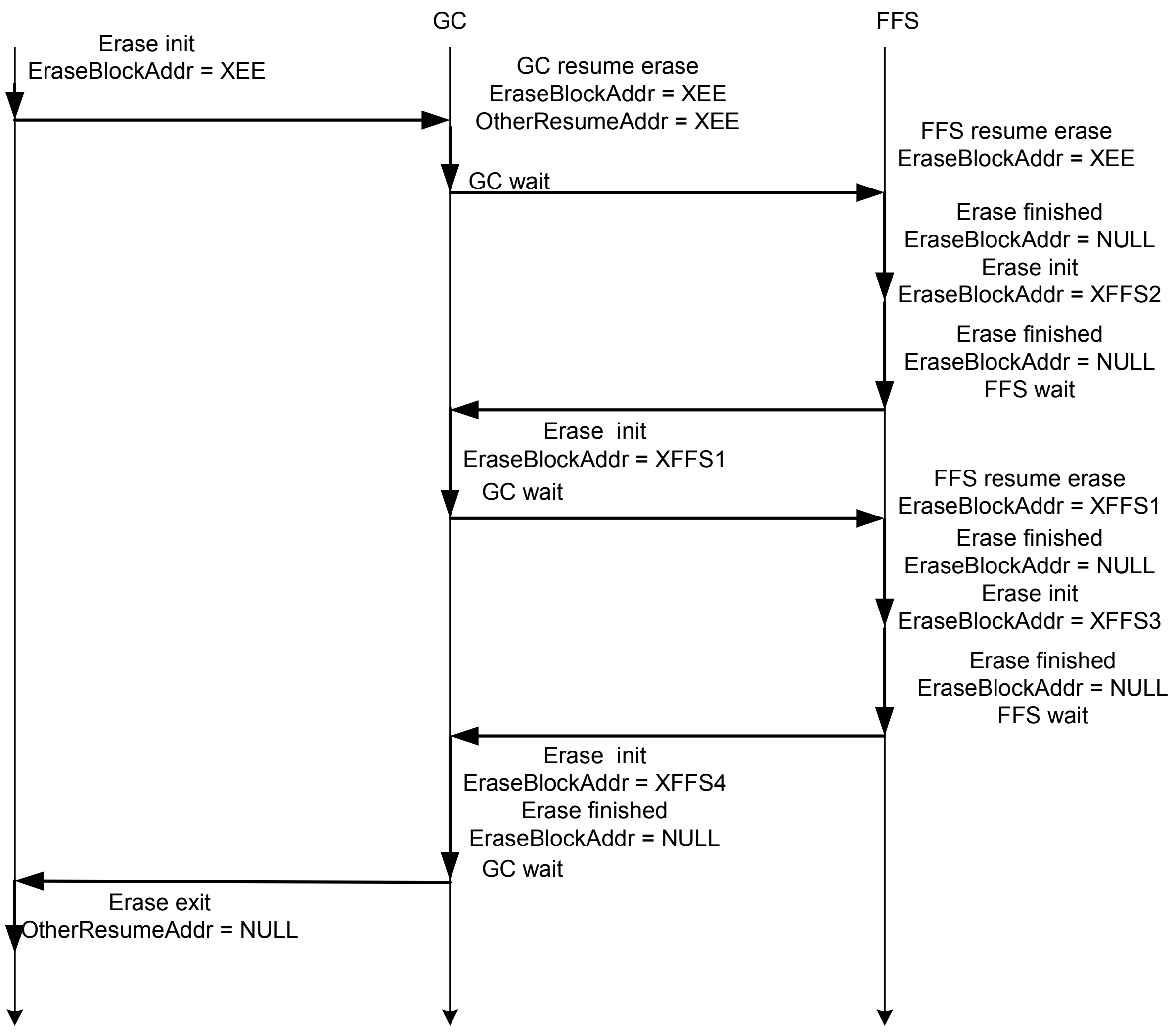
With the FFS LFA and forced GC the FFS task has become a task with the ability to erase a block. Previously only the Garbage Collection (GC) task and the EE-idle (EEI) task had that feature. An erase of a block must be finished before an erase of another block is initiated. Because erase suspend is used to secure handle of interrupts the code must be able to handle that another higher prioritized task can take over the erase. This has been handled with the consideration that higher prioritized tasks must be able to schedule during an erase but the previously initiated erase has to be completed. The active task must complete an erase that have been initiated by a lower prioritized task before another block can be erased.

The real time erase design has the following preconditions:

- The 3 tasks only erase blocks that the other tasks aren't erasing. GC erases the FFS blocks, FFS erases LFA and FFS blocks and EEI erases the EE blocks. FFS only erases the FFS blocks taking precaution of if GC is erasing a FFS block when forced GC is in use.
- The tasks are prioritized from lowest as EEI, GC and FFS. FFS is the highest prioritized task.
- (ONLY nucleus systems = SGOLD) EEI is the only task that has the peculiarity that it restarts the erase procedure from the beginning of the erase procedure instead of where it came from when it returns from an interrupt. To avoid a new erase after another task have finished the erase the variable OtherResumeAddr have been added.

To make a task be able to see if another erase must be resumed before an erase can be initialized the variable EraseBlockAddr have been added. EraseBlockAddr is set when an erase is initialized and deleted when the erase is finished.

The prioritization makes the worst case appear when EEI has initiated an erase and is interrupted. The GC is scheduled and wants to initiate an erase. GC resumes the previously initiated erase and is then scheduled to wait. The FFS is scheduled and wants to initiate an erase. It finishes the previously initiated erase, initiates its own erase and finishes. The worst case is shown in the figure.



Notice that OtherResumeAddr is only written when an erase is resumed and OtherResumeAddr have been reset. It is only reset when the task that initiated the erase exits the erase procedure and at initialisation.

2.17.2 Multi tasking at programming

Just like the multi tasking issue at erase, the same problem can occur when the flash is programmed and program suspend is used to secure interrupt handles. Also the same conditions are present for programming as for erase. Multi tasking programming has been handled the same way as erase.

2.18 Power up.

2.18.1 Power Loss Recovery.

If power loss happens by accident while any flash modifying operation is in progress, it should be possible to detect this at the next power up.

Possible situations to detect:

a). A file-write operation was in progress when powered off.

At power up the PLR algorithms will look through all valid files and register the sectors used by the files.
This will be matched against a complete search for used sectors, if any sectors are found in use but not as a part of a valid file they will be marked as invalid and the GC will clean them up.

b). A block clean up by the garbage collector was in progress.

The recovery method is here dependent of how far the garbage collector was in the cleanup process. By examining the header control bits in the GI and GC blocks, the state before power loss can be found, and the clean up can be resumed.

The different situations are described below.

1) No GC block is found.

The old GC block has been erased, and a new one has not yet been allocated.

A new GC block will be allocated.

2) The BIU bit is cleared, but the GCV field is not updated in the GC block.

The garbage collector has initiated a clean of a block but the power was lost before all valid data was copied to the GC block.

The following will be done:
- Clear the RFE bit in the GC block.
- Erase the GC block.
- The GC block BEC field is incremented.
- A new GC block will be allocated.

3). The GC block has the RFE (Ready For Erase) bit cleared.

An error has happened and the garbage collector has aborted a cleanup.

The following will be done:

- Erase the GC block.
- The GC block BEC field is incremented.
- A new GC block will be allocated.

4). The GC block has GCV cleared, the CBN and LBN is updated and the GI block has the same LBN, but the GCC is not cleared.

All valid data has been copied to the GC block, but the GI block hasn't been erased yet.

The following will be done:
- Erase the GI block.
- The GI block BEC field is incremented.
- Clear the GCC bit in the GC block.
- A new GC block will be allocated.

5). The GC block has GCV cleared, but not the GCC, and GI has a different LBN as the GC block

All valid data has been copied to the GC block and the GI block has been erased but the GCC bit needs to be cleared in the GC block.

The following will be done:
- Clear the GCC bit in the GC block.
- A new GC block will be allocated.

2.18.2 Configuration at power up.

At power up the FFS system will configure itself by examining the contents of the FFS flash store.

The following will be configured:

- File Allocation Table (FAT)

The FFS area in the flash is searched, and all the files contained are located and registered in the FAT.
The FAT start sector field is initialized, and file state is set to *closed*.

The PIT (Pre-assigned Id Table) is searched. For pre-assigned id's with no physical file stored in the FFS, the FAT file state will be set to *assigned.*

- Memory Use Table (MUT)

During the search for valid files, FFS will calculate the memory consumption of each found file and update the MUT. The total memory consumption and the individual values for each memory pool are calculated.

- Sector_usage Table.

FFS will update the sector_usage table with the number of used sectors and the number of invalid sectors in each block.

This is used by the Garbage Collector to determine whether clean up is needed or not.

- HW_logic_blocks Table.

In each block, which is marked as in use, the logical block number is read and the HW_logic_blocks table is updated with this information.

2.18.3 Factory default files at power up.

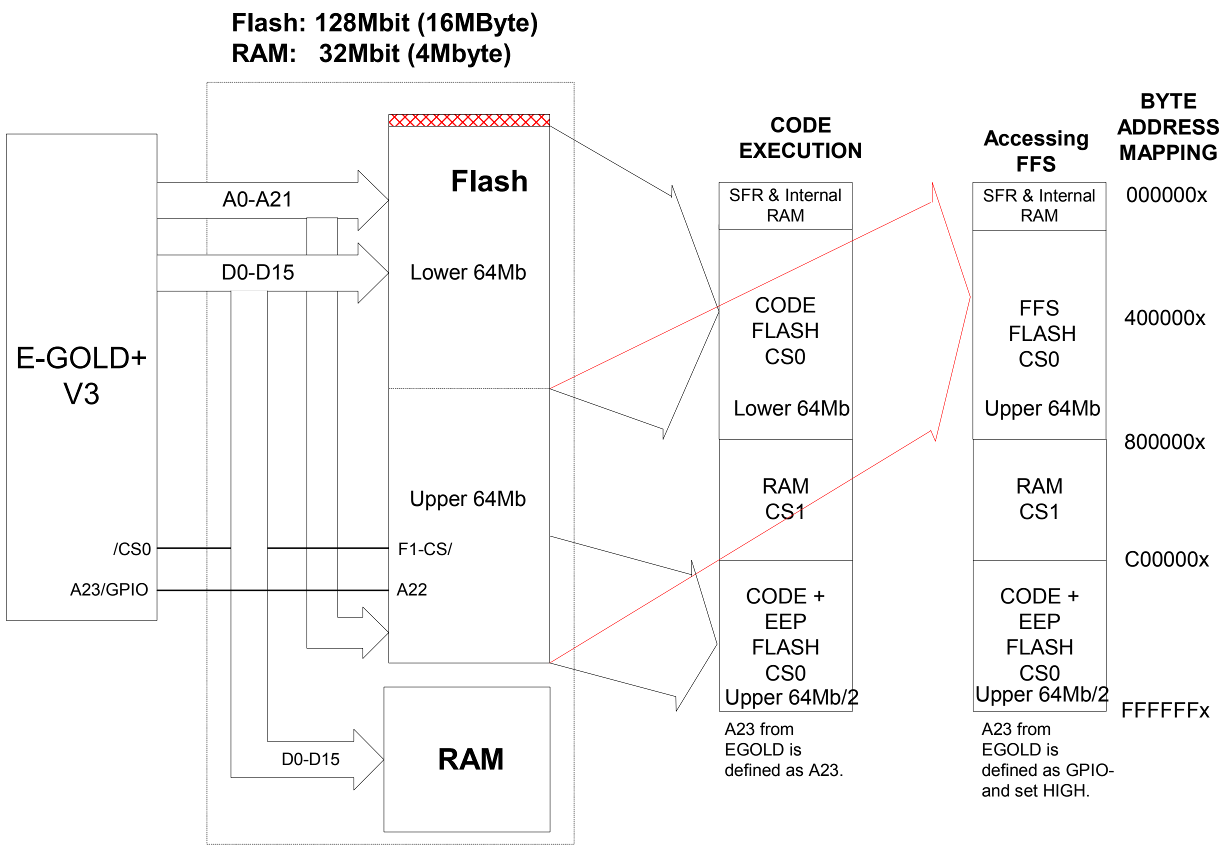
At power up the FFS will locate all files stored in FFS. If it find a file marked as a factory default file it will check if the dynamic duplicate exists. If not it will create it by making a copy of the static factory default file.

2.19 Bank Switch Concept

The E-gold chip has a limited address range of 128 Mbit and in large SW solution this can be to small to include Flash Code, RAM and Flash to store FFS data. The bank switch concept hides the FFS data storage behind the RAM area when FFS isn't accessed and switches it to a visible position when the FFS data shall be accessed. Currently there are defined two types of bank switch.

2.19.1 CS0 Bank switch

With the CS0 concept there is used one flash on 128 Mbit on chip select 0 and one 32 Mbit RAM on CS1.

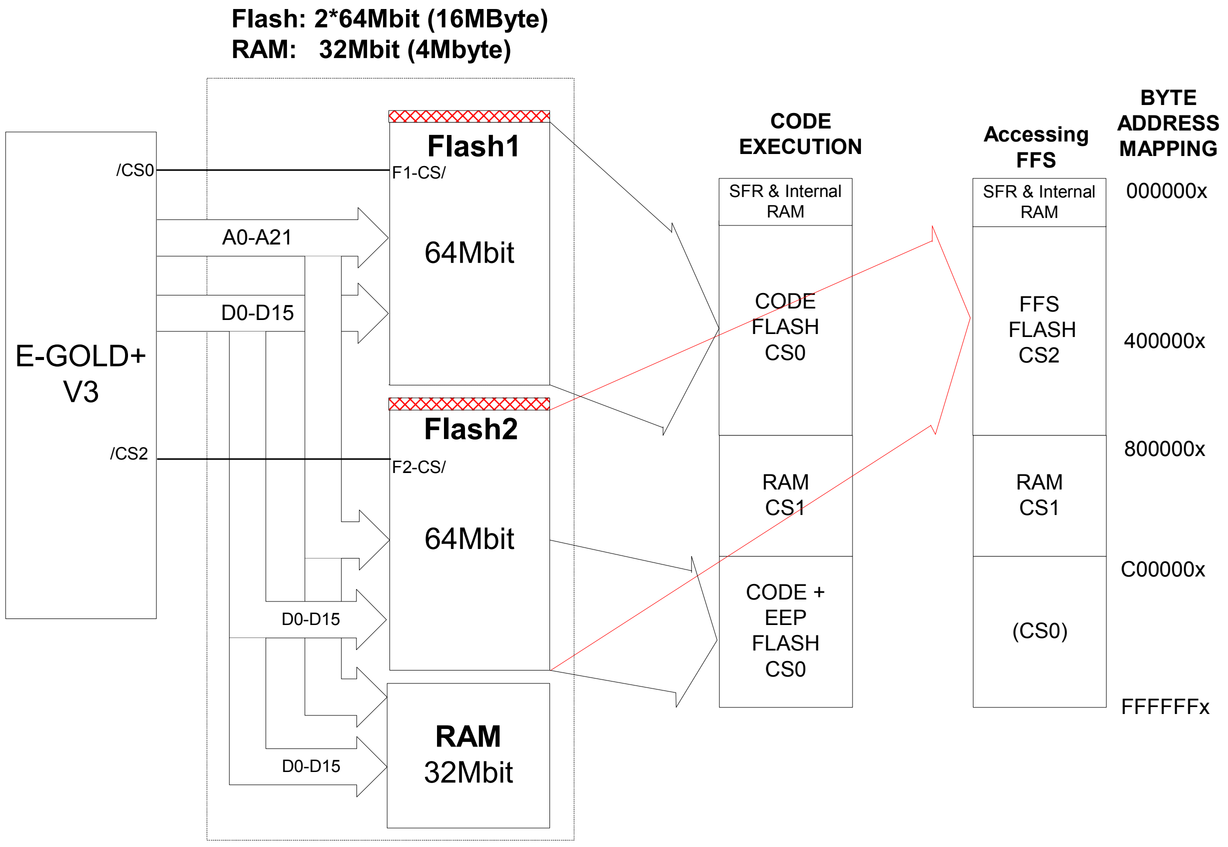


With this concept, it is possible to have up to 96 Mbit CODE flash and up to 64 Mbit DATA flash. Max. 128 Mbit in total in CODE + RAM and CODE + DATA.

When FFS wants to access the data area it set A23 permanently high and thereby switches the high part of the flash to have the same address mapping as the lower half. The data can then by accessed with the addresses from 0x000000 – 0x7FFFFFF and the RAM area can be avoided.
When using this concept it is important to place modules on CS1, CS2, CS3 and CS4 on address spaces outside the switched FFS data area or disable these chip selects when accessing FFS data because CS0 has the lowest CS priority.

2.19.2 CS2 Bank switch

With the CS2 concept there is used two flashes on 64 Mbit on chip select 0 & 2 and one 32 Mbit RAM on CS1.



With this concept, it is possible to have up to 96 Mbit CODE flash and up to 64 Mbit DATA flash. Max. 128 Mbit in total in CODE + RAM and CODE + DATA.

When FFS wants to access the data area it changes the definition of CS2 to be active in the address space from 0x000000 – 0x7FFFFFF and thereby switches the flash 2 to have the same address mapping as flash 1. The data can then by accessed with the addresses from 0x000000 – 0x7FFFFFF and the RAM area can be avoided.
When using this concept it is important to place modules on CS1 on address spaces outside the switched FFS data area or disable these chip selects when accessing FFS data because CS1 has higher CS priority than CS2.

3 Customization of FFS

FFS can be customized by adjusting the parameters in the ffs_def.h file. The file consist of a number of defines, enumerations and arrays which can be customized. FFS must be recompiled after customization.

In the file, there is a revision number, which must be incremented if the parameters are changed.

3.1 Customizable defines

Define	Description
FFS_DEF_REVISION	This is a revision number which must be incremented if any of the parameters are changed
FFS_BLOCK_SIZE	This is the physical size on one block on the flash. This value is given by the manufacture of the flash
FFS_SECTOR_SIZE	Each block is divided into a number of sectors and this is the size of each sector, see also chapter 2.11. Fixed to the region size.
FFS_NOF_STATIC_BLOCKS	This is the number of flash blocks used to the static part of FFS, see also chapter 2.8
FFS_NOF_DYNAMIC_BLOCKS	This is the number of flash blocks used to the dynamic part of FFS, see also chapter 2.8
FFS_BLOCK_ADDRESSES	This is an array of start address of the flash blocks to use. There must be one entry for each static and each dynamic block. See also chapter 2.10.1
FFS_MAX_NOF_FILES	This is the maximum number of files, which can be stored in FFS at the same time.
FFS_LFA_ADDR_NOF_BLOCKS	This is a two dimensional array with the flash blocks to use if LFA is included, see also chapter 2.16. Each entry must include the start address and the number of consecutive blocks to use for each LFA sector.
FFS_MAX_SOF_FILENAME	If the filename feature is used this defines the maximum size of the filename including NULL termination.
FFS_MAX_SOF_DIR_NAME	If the filename and directory support feature is used this defines the maximum size of the directory name including NULL termination.
FFS_COPY_TEMP_MAX_SIZE	This defines how many bytes can be copied in a copy operation, before FFS must check if there are any higher priority user request.
FFS_LFA_TEMP_MAX_SIZE	This defines how many bytes can be copied in a copy to LFA operation, before FFS must check if there are any higher priority user request.

FFS_SIBLEY_REGION_SIZE	This is the region size that FFS will use as the sector size. For a segmented region type flash like the Sibley flash the region must be fixed to follow the flash regions, See chapter 1.8.
FFS_SIBLEY_SEGMENT_SIZE	The segment size determines the segmentation of File Start Sectors, Interface Sectors, the block sector and the sector header sectors. See chapters 1.8 and 2.11. The segment size must be a size that can be divided with the region size and give a whole number.
FFS_MAX_NOF_STREAMING_FILES	The numbers of streaming files that are open at the same time for read or write. For every open streaming file there is a streaming buffer attached on a sector size.
FFS_MAX_NOF_WRITE_BUFFERS	The numbers of write buffers available for write when a segmented region type flash is used. The buffers are on a sector size each. The write buffers are used to avoid write modify.
FFS_MAX_FILE_SI ZE	The maximum file size that it is possible to write to FFS.
FFS_MAX_NOF_DIRS	The maximum number of directories at FFS.
FFS_FS_IF_SECTOR_SIZE	The size of a File Start sector or Interface Sector.

3.2 Customizable enums

3.2.1 USER ID TABLE

This enum contains an element for each process using the FFS. FFS uses this table to priorities multiple operations requested by different tasks. The first element has the highest priority and the last element has the lowest priority. The elements 'FFS_DWDIO', 'FFS_ATC' and 'FFS_NOF_USERS' shall always be defined. See also chapter 2.10.8

```
typedef enum {
  FFS_DWDIO,

 ‘Place new users here’

  FFS_ATC,
  FFS_NOF_USERS      //Dummy element
  } ffs_user_type;
```

3.2.2 PIT (Pre-assigned Id Table)

In the PIT it's possible to reserve some file-id's at compile-time. This can be used for files that are downloaded during production. See also chapter 2.10.7

```
enum ffs_pit {
 'Place new id's here'

  FFS_NOF_RES_ID   //Dummy element
};
```

3.2.3 FILE TYPES

The files in FFS is divided into a number of filetypes which each have a dedicated memory pool. See also chapter 2.9.1, 2.10.2 and 2.10.3. Remember to update "ffs_memory_pool_reservations" when changing this.

```
typedef enum {
  FFS_GENERIC_FILETYPE,        // For files with no specific type, or for temp-files.

 'Place new files here'

  FFS_NOF_FILETYPES,
  FFS_FILETYPE_GARBAGE,        // Special filetype for garbage collector ONLY.
  FFS_ALL_FILETYPES = 0xFF
} ffs_filetype_type;
```

3.3 Customizable arrays

3.3.1 FFS MEMORY POOL RESERVATIONS

This array defines how much memory there shall be dedicated to each filetype. All unreserved memory will be allocated for the global pool. As a rule of thumb, at least 20% of the total memory should be reserved for the global pool since this is the only space FFS can use as work space. The different file types must be listed in the same order as in the "ffs_filetype_type".
Only dynamic memory can be reserved.
To avoid allocating a new array for each module including the ffs_def.h this array can be encapsulated with the FFS_INCLUDING_FILE compiler define.

```
#ifdef FFS_INCLUDING_FILE
const ulong ffs_memory_pool_reservations[FFS_NOF_FILETYPES] = {
 /* FFS_GENERIC_FILETYPE                */   0,

'Place new files here'

};
#endif // FFS_INCLUDING_FILE
```

4 References

[1]	Title:	EEPROM specification
	Author:	Erik Christensen
	Revision:	0.8
[2]	Title:	FFS Interface specification Flash File System
	Author:	Erik Christensen
	Revision:	1.6
[3]	Title:	FFS PC interface
	Author:	Erik Christensen
	Revision:	1.6
[4]	Title:	Introduction to Storage Tool
	Author:	Rasmus Kaae
	Revision:	1.1
[5]	Title:	Generic Module Test System User Guide
	Author:	Suresh Bhaskaran
	Revision:	5.0
[6]	Title:	FlashTool Download and Update application
	Author:	Niels Jørgen Kofod
	Revision:	4.1
[7]	Title:	Flash Endurance Simulator, Design
	Author:	Prasanth Ambalakundu Gangadharan
	Revision:	1.0
[8]	Title:	Flash Endurance Simulator, User Guide
	Author:	Prasanth Ambalakundu Gangadharan
	Revision:	1.0

5 Appendix

5.1 Estimation of Wear Leveling parameters

5.1.1 Testing wear leveling

Wear leveling has been tested with simulation using the FES tool. With the FES tool, it is possible to simulate the use of a flash over a flash lifetime but in “a very short time”. Read more about the FES tool at [7] and [8]. 3 examples has been used:

- Wear test example: 17 blocks with static data and 29 blocks with different dynamic data.
- Dynamic test example: All blocks contain dynamic data but some of the data parts are so large that blocks with “static” data occur temporarily.

Pure dynamic test example: Only very dynamic data with small data parts occurs in the test

The tests have a certain amount of build in randomness. The tests are stopped when one flash block reaches the maximum lifetime. The maximum lifetime are chosen to 10000.

5.1.2 Wear Leveling design issues

Wear leveling is an operation used with the flash file system to “wear out” the use of the blocks in the flash. A flash only has a limited lifetime where it is possible to erase a block a number of times, i.e. 100000 times. The ideal use of the flash in a flash file system is to erase all blocks the same number of times. This is often not possible because there can be data parts that are changed very seldom. With wear leveling, a block that hasn’t been erased as much as other blocks, because it contains data that are changed seldom, are moved to another block that has been erased often. The old block is erased for future use.

5.1.3 Decision parameters for wear leveling

5.1.3.1 Difference between average and minimum BEC decision level

The decision criteria’s for wear leveling are set with a max and a min for the difference between the average and minimum BEC, with a decision level parameter as the linear decreasing parameter between these limits over a lifetime level.

The following tables show the result of a FES test with the wear test example where max is 300 and min is 1. The lifetime for linearity is set to 10000. The level indicates the actual threshold value. The table shows the result of the test example when the test is stopped to check out data with the decrease of the linearity level by 10.

Wear Leveling test example, max= 300, min=1, linear lifetime 10000, choice=0

Days	difference in days	min	average	difference average	Max	no wear levelings	difference no wl	level	variance	erases
3		0	1		3	0		300	1	38
242	239	0	198	197	335	0	0	290	155	4937
500	258	188	414	216	669	41	41	280	156	10354
786	286	430	663	249	1003	41	0	270	148	16363
1078	292	704	930	267	1337	70	29	260	170	22484
1416	338	995	1241	311	1672	89	19	250	152	29597
1772	356	1359	1571	330	2006	108	19	240	139	37051
2148	376	1700	1922	351	2340	125	17	230	121	44963
2493	345	2030	2245	323	2675	151	26	220	126	52208
3024	531	2542	2743	498	3010	184	33	210	131	63363
3330	306	2859	3042	299	3343	208	24	200	125	69789
3641	311	3153	3342	300	3679	239	31	190	124	76319
3972	331	3485	3664	322	4013	262	23	180	108	83270
4274	302	3789	3959	295	4347	283	21	170	117	89599
4617	343	4137	4296	337	4682	311	28	160	93	96802
5161	544	4688	4838	542	5016	347	36	150	80	108235
5477	316	5019	5155	317	5351	383	36	140	90	114863
5783	306	5327	5456	301	5684	411	28	130	79	121302
6092	309	5671	5760	304	6020	448	37	120	66	127781
6446	354	6007	6114	354	6354	500	52	110	70	135215
6776	330	6339	6440	326	6689	542	42	100	63	142135
7262	486	6831	6920	480	7023	610	68	90	48	152358
7594	332	7174	7248	328	7357	671	61	80	41	159307
7930	336	7512	7583	335	7692	735	64	70	41	166376
8260	330	7853	7912	329	8025	797	62	60	36	173294
8611	351	8217	8269	357	8361	884	87	50	34	180691
8905	294	8525	8566	297	8694	977	93	40	28	186849
9300	395	8937	8968	402	9031	1132	155	30	18	195160
9621	321	9283	9302	334	9364	1320	188	20	15	201888
9976	355	9659	9669	367	9698	1628	308	10	6	209344
10276	300	9975	9983	314	10000	2127	499	1	5	215647

It shows from the table that the number of wear leveling increases when the level decreases. The variance improves with a larger number of wear leveling.

Level	Difference no wl / difference average [%]	no wl / average	difference no wl / no wl	no wl / erases
300				
290	0,00%	0,00%		0,000%
280	18,98%	9,90%	100,00%	0,396%
270	0,00%	6,18%	0,00%	0,251%
260	10,86%	7,53%	41,43%	0,311%
250	6,11%	7,17%	21,35%	0,301%
240	5,76%	6,87%	17,59%	0,291%
230	4,84%	6,50%	13,60%	0,278%
220	8,05%	6,73%	17,22%	0,289%
210	6,63%	6,71%	17,93%	0,290%
200	8,03%	6,84%	11,54%	0,298%
190	10,33%	7,15%	12,97%	0,313%
180	7,14%	7,15%	8,78%	0,315%
170	7,12%	7,15%	7,42%	0,316%
160	8,31%	7,24%	9,00%	0,321%
150	6,64%	7,17%	10,37%	0,321%
140	11,36%	7,43%	9,40%	0,333%
130	9,30%	7,53%	6,81%	0,339%
120	12,17%	7,78%	8,26%	0,351%
110	14,69%	8,18%	10,40%	0,370%
100	12,88%	8,42%	7,75%	0,381%
90	14,17%	8,82%	11,15%	0,400%
80	18,60%	9,26%	9,09%	0,421%
70	19,10%	9,69%	8,71%	0,442%
60	18,84%	10,07%	7,78%	0,460%
50	24,37%	10,69%	9,84%	0,489%
40	31,31%	11,41%	9,52%	0,523%
30	38,56%	12,62%	13,69%	0,580%
20	56,29%	14,19%	14,24%	0,654%
10	83,92%	16,84%	18,92%	0,778%
1	158,92%	21,31%	23,46%	0,986%

When the difference in the number of wear leveling is compared with the difference in the average BEC it figures that the relation between the number of wear leveling and average BEC seems to be stable between 0-14 % until level 90. Then the numbers of wear leveling in relation to average BEC increases.

When the difference in the number of wear leveling and the number of wear leveling are compared it figures that the number of wear leveling are large at the beginning decreasing to a minimum around level 130 – 60 and then increasing again.

When the level decreases below 30 the number of wear leveling accelerates significantly.

When the number of wear leveling are compared to the number of erases it figures that wear leveling gives a overhead in erases below 1 %.

The following values have been chosen
Max = 100
Min = 30
Lifetime = 100000 (should support all flashes)

The maximum also determines the latency before wear leveling operations begins because this is the initial condition level between the minimum and average BEC.

5.1.3.2 The static data decision criteria

The other decision criteria for wear leveling is the "static data criteria" given by the "same choice" parameter. The "same choice" parameter gives the number of times where the same flash block is chosen to be the one to "wear out".

The "same choice" parameter has been tested with different values with the Wear and Dynamic test example. The tables below show the test results when the wear test example and the dynamic test example are run until a maximum at 10000 (despite that the lifetime level is set to 100000) with different choices of the "same choice" parameter. The test results shows that the Dynamic test doesn't contain real static data, but data that are static for a period. When the "same choice" parameter is larger or equal to 10 then no wear leveling are executed. However, the dynamic test also shows that wear leveling helps getting a smaller variance and a larger minimum BEC value. The Dynamic test also shows that if the "same choice" parameter is very small then the number of wear leveling operations are increased significantly.

Wear test example

same choice >	lifetime level	days	min	average	no wl	variance	erases
0	10000	9920	9884	9914	2339	18	208139
1	10000	10001	9923	9954	2122	19	209849
2	10000	10134	9892	9922	2138	22	212652
3	10000	9987	9847	9888	2132	30	209567
4	10000	10209	9855	9898	2041	26	214213
5	10000	9984	9841	9895	1867	34	209472
10	10000	9838	9684	9798	1338	64	206404
0	100000	9803	9708	9800	1461	56	205650
1	100000	9965	9736	9826	1427	56	209076
2	100000	10017	9743	9835	1291	53	210200
3	100000	10255	9723	9817	1439	64	215171
4	100000	10104	9793	9880	1341	53	211999
5	100000	10177	9721	9811	1421	59	213568
10	100000	9854	9714	9822	1141	60	206952

Dynamic test example

same choice >	lifetime level	days	min	average	no wl	variance	erases
0	10000	1977	9701	9746	5376	40	219344
1	10000	1936	9286	9533	1414	112	214792
2	10000	1906	8864	9360	373	173	211451
3	10000	1968	8991	9692	99	180	218346
4	10000	1969	8885	9694	22	177	218452
5	10000	1945	8804	9565	12	202	215783
10	10000	1963	8911	9620	0	177	217795
0	100000	1948	9536	9634	3392	82	216133
1	100000	1969	9378	9668	1484	116	218454
2	100000	1959	9139	9614	408	178	217348
3	100000	1939	8864	9509	85	179	215121
4	100000	1948	8962	9536	26	173	216114
5	100000	1938	8825	9503	14	204	215021
10	100000	1933	8770	9483	0	181	214450

The “same choice” parameter has been chosen to be 2 to get wear leveling fast but not too often.

5.1.4 Wear leveling test results

With the chosen wear leveling parameters the test examples gives the following results.

5.1.4.1 The pure dynamic test example

This test can’t initiate wear leveling operations. Ordinary garbage collection makes an even use of all the dynamic blocks.

5.1.4.2 The Wear leveling test example

Wear test example

	days	min	average	max	no wl	erases
With wear leveling	10017	9743	9835	10001	1291	210200
Without wear leveling	7230	0	6969	10000	0	151651

5.1.4.3 The Dynamic test example

Dynamic test example

	days	min	average	max	no wl	erases
With wear leveling	1959	9139	9614	10003	408	217348
Without wear leveling	1924	8800	9505	10003	0	213459

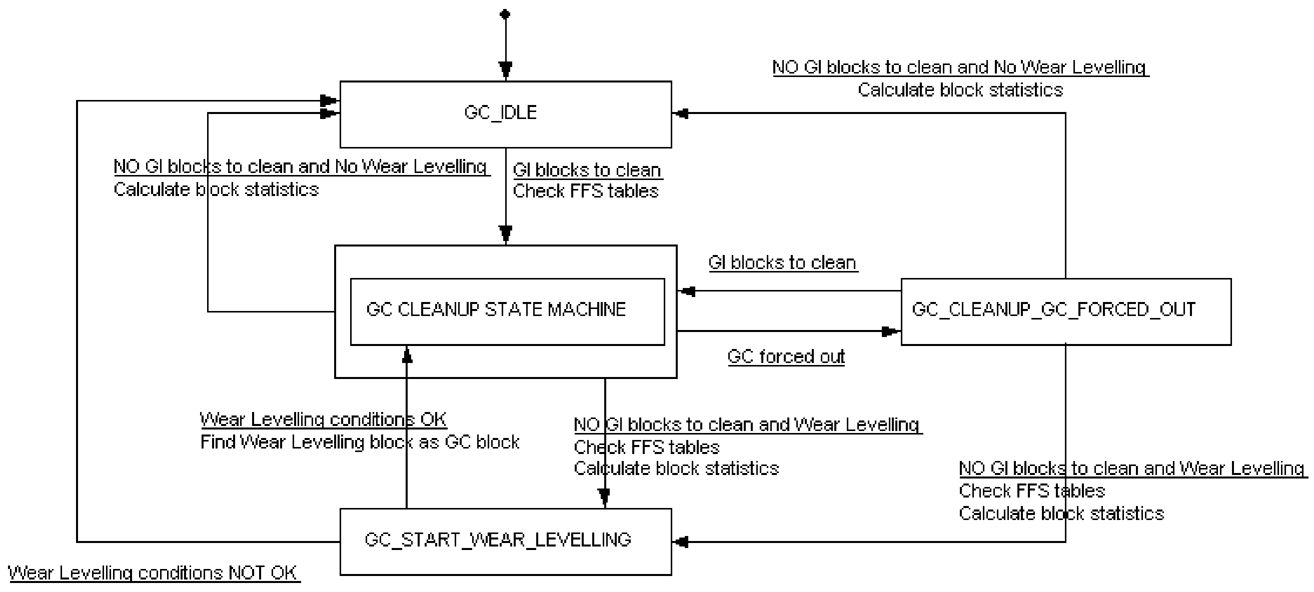
5.2 GC state

The following GC state machines describes GC operations. Events are written with an underline and actions are written after the belonging event.

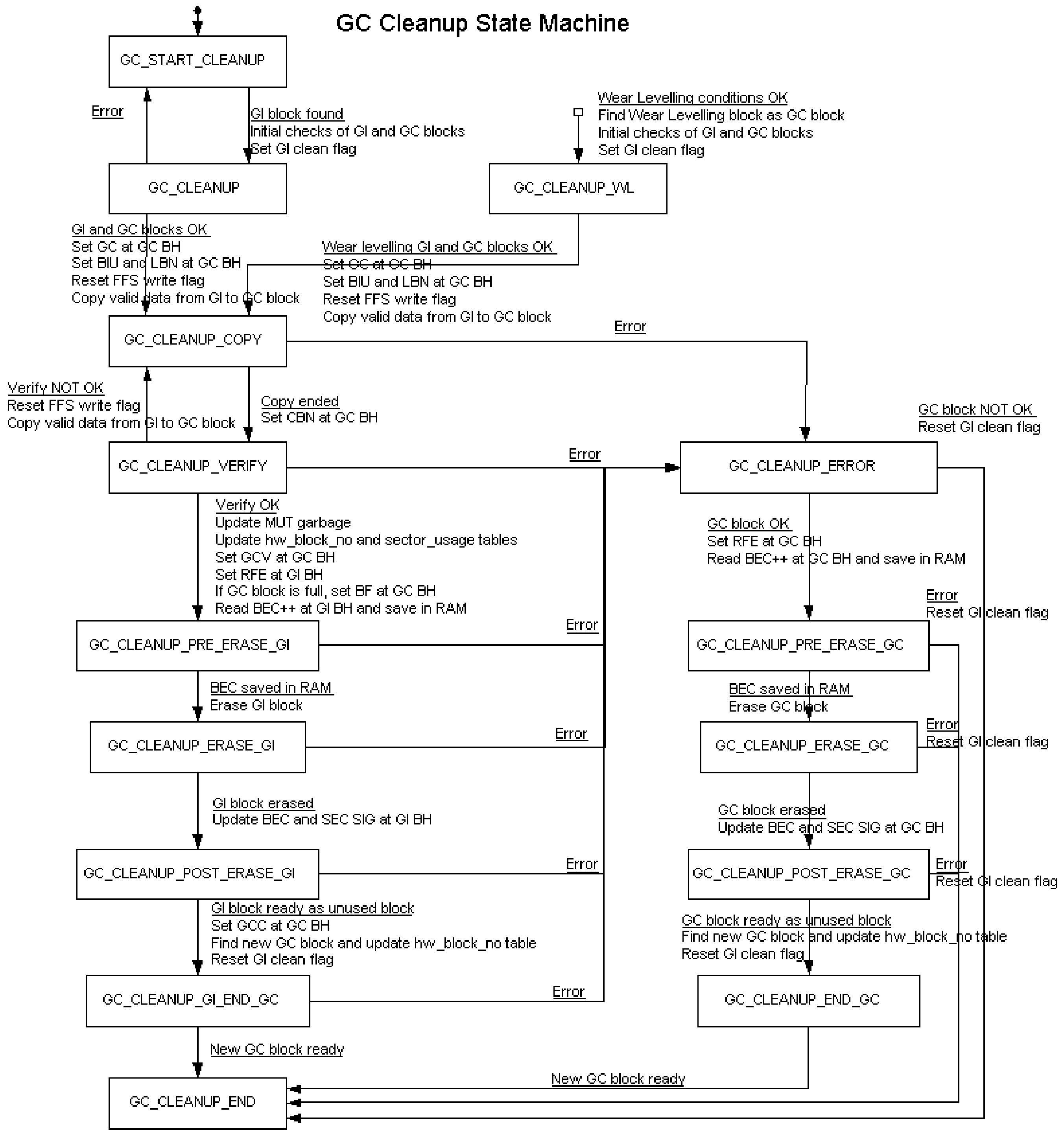
5.2.1 GC state machine

In the shown state machine "GC CLEANUP STATE MACHINE" is a super state. The super state is shown below.

GC State Machine


GC_IDLE
GC CLEANUP STATE MACHINE
GC_CLEANUP_GC_FORCED_OUT
GC_START_WEAR_LEVELLING
NO GI blocks to clean and No Wear Levelling
Calculate block statistics
NO GI blocks to clean and No Wear Levelling
Calculate block statistics
GI blocks to clean
Check FFS tables
GI blocks to clean
GC forced out
Wear Levelling conditions OK
Find Wear Levelling block as GC block
NO GI blocks to clean and Wear Levelling
Check FFS tables
Calculate block statistics
NO GI blocks to clean and Wear Levelling
Check FFS tables
Calculate block statistics
Wear Levelling conditions NOT OK

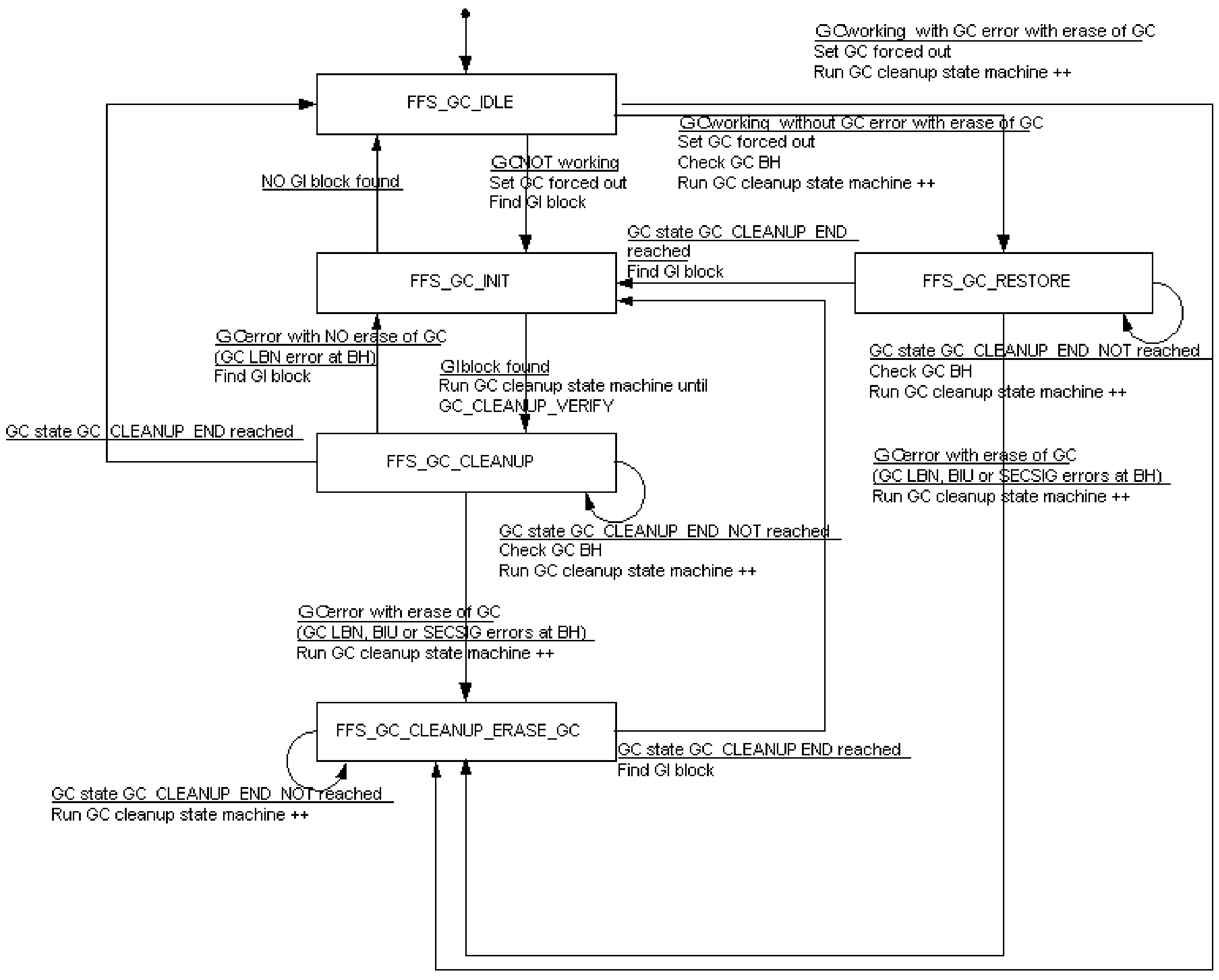
5.2.2 GC Cleanup State machine



5.3 FFS forced GC

5.3.1 FFS GC state machine

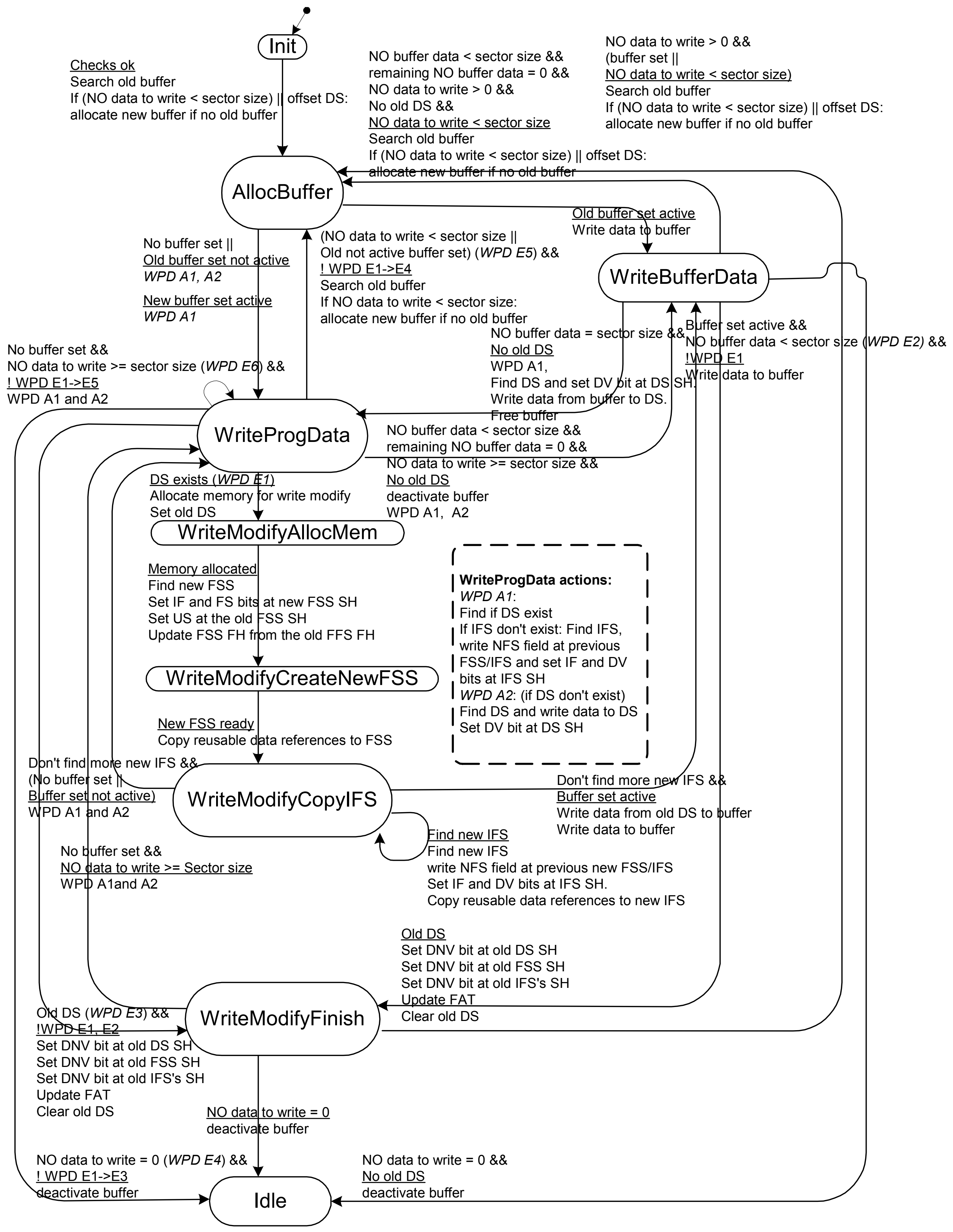
FFS forced GC State Machine



FFS_GC_IDLE
FFS_GC_INIT
FFS_GC_RESTORE
FFS_GC_CLEANUP
FFS_GC_CLEANUP_ERASE_GC
GC working with GC error with erase of GC
Set GC forced out
Run GC cleanup state machine ++
GC working without GC error with erase of GC
Set GC forced out
Check GC BH
Run GC cleanup state machine ++
GC NOT working
Set GC forced out
Find GI block
NO GI block found
GC state GC_CLEANUP_END reached
Find GI block
GC error with NO erase of GC
(GC LBN error at BH)
Find GI block
GI block found
Run GC cleanup state machine until
GC_CLEANUP_VERIFY
GC state GC_CLEANUP_END NOT reached
Check GC BH
Run GC cleanup state machine ++
GC state GC_CLEANUP_END reached
GC state GC_CLEANUP_END NOT reached
Check GC BH
Run GC cleanup state machine ++
GC error with erase of GC
(GC LBN, BIU or SECSIG errors at BH)
Run GC cleanup state machine ++
GC error with erase of GC
(GC LBN, BIU or SECSIG errors at BH)
Run GC cleanup state machine ++
GC state GC_CLEANUP END reached
Find GI block
GC state GC_CLEANUP_END_NOT reached
Run GC cleanup state machine ++

5.4 Write with write buffers

5.4.1 Write with write buffers state event diagram


Init
Checks ok
Search old buffer
If (NO data to write < sector size) || offset DS:
allocate new buffer if no old buffer
NO buffer data < sector size &&
remaining NO buffer data = 0 &&
NO data to write > 0 &&
No old DS &&
NO data to write < sector size
Search old buffer
If (NO data to write < sector size) || offset DS:
allocate new buffer if no old buffer
NO data to write > 0 &&
(buffer set ||
NO data to write < sector size)
Search old buffer
If (NO data to write < sector size) || offset DS:
allocate new buffer if no old buffer
AllocBuffer
Old buffer set active
Write data to buffer
No buffer set ||
Old buffer set not active
WPD A1, A2
New buffer set active
WPD A1
(NO data to write < sector size ||
Old not active buffer set) (WPD E5) &&
! WPD E1->E4
Search old buffer
If NO data to write < sector size:
allocate new buffer if no old buffer
WriteBufferData
NO buffer data = sector size &&
No old DS
WPD A1,
Find DS and set DV bit at DS SH.
Write data from buffer to DS.
Free buffer
Buffer set active &&
NO buffer data < sector size (WPD E2) &&
!WPD E1
Write data to buffer
No buffer set &&
NO data to write >= sector size (WPD E6) &&
! WPD E1->E5
WPD A1 and A2
WriteProgData
NO buffer data < sector size &&
remaining NO buffer data = 0 &&
NO data to write >= sector size &&
No old DS
deactivate buffer
WPD A1, A2
DS exists (WPD E1)
Allocate memory for write modify
Set old DS
WriteModifyAllocMem
Memory allocated
Find new FSS
Set IF and FS bits at new FSS SH
Set US at the old FSS SH
Update FSS FH from the old FFS FH
WriteProgData actions:
WPD A1:
Find if DS exist
If IFS don't exist: Find IFS,
write NFS field at previous
FSS/IFS and set IF and DV
bits at IFS SH
WPD A2: (if DS don't exist)
Find DS and write data to DS
Set DV bit at DS SH
WriteModifyCreateNewFSS
New FSS ready
Copy reusable data references to FSS
Don't find more new IFS &&
(No buffer set ||
Buffer set not active)
WPD A1 and A2
WriteModifyCopyIFS
Don't find more new IFS &&
Buffer set active
Write data from old DS to buffer
Write data to buffer
Find new IFS
Find new IFS
write NFS field at previous new FSS/IFS
Set IF and DV bits at IFS SH.
Copy reusable data references to new IFS
No buffer set &&
NO data to write >= Sector size
WPD A1and A2
Old DS
Set DNV bit at old DS SH
Set DNV bit at old FSS SH
Set DNV bit at old IFS's SH
Update FAT
Clear old DS
WriteModifyFinish
Old DS (WPD E3) &&
!WPD E1, E2
Set DNV bit at old DS SH
Set DNV bit at old FSS SH
Set DNV bit at old IFS's SH
Update FAT
Clear old DS
NO data to write = 0
deactivate buffer
NO data to write = 0 (WPD E4) &&
! WPD E1->E3
deactivate buffer
Idle
NO data to write = 0 &&
No old DS
deactivate buffer