BNG Bootloader

0.2

Generated by Doxygen 1.6.3

Tue Apr 20 02:11:45 2010

Contents

Chapter 1

API Specification

This document contains the whole API specification for the BNG Bootloader project, as generated by the Doxygen documentation generator tool.

The document is built from the C sources into different formats, it's made available in HTML and PDF formats. Please refer to the project's documentation for the exact location of these documents.

The complete project (prosa) documentation is built from ReST sources using Sphinx and can be found in HTML format at the address: `http://garetjax.github.com/Bootloader/`

1.1 Overview

Todo

Move this documentation to another file or to the Sphinx documentation.

Before presenting the in-depth API documentation, the UML diagram in Fig. 1.1 was created to present an higher-level overview of the API.

The `Controller` "class" representes our `main` function, and is responsible for the initialization of the various parts of the system, their respective interconnection (serial interface initialization, command registration,...) and the routing of every received command to the command dispatcher, which will in turn call the right command passing the parsed tokens as arguments.

Each command will then execute its task interacting with either the application manager or the configuration registry.

Note

- Different realation graphs are automatically generated by Doxygen during the build phase, please refer to them for an in-dept view.
- In the diagram below, each package representes a component, a class a C source file and the methods, the different functions contained in the file (the only exception is the `Command` interface, which representes the type of a command like function).
- The details about the function prototypes where omitted for the sake of clarity.

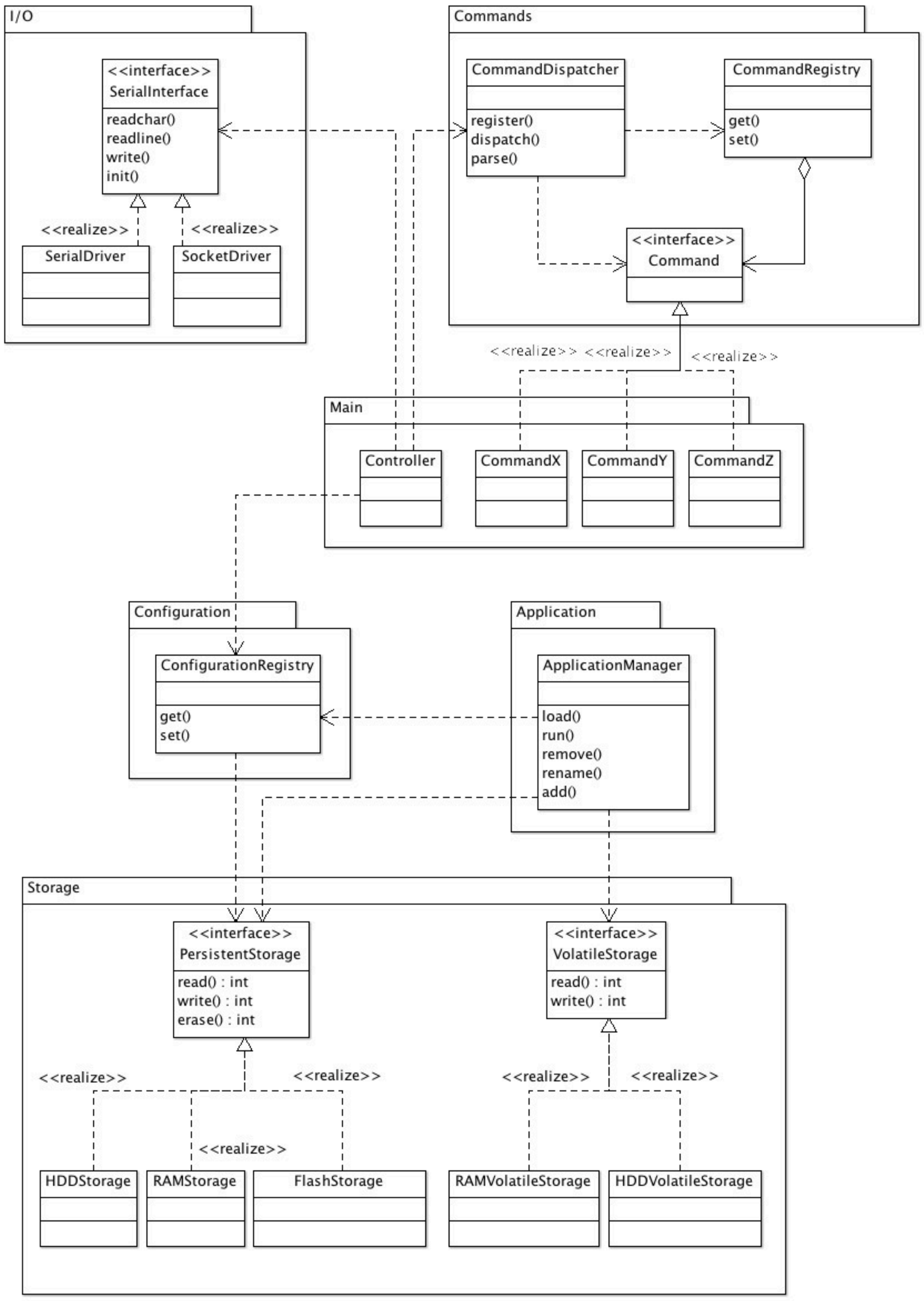


Figure 1.1: API Overview

Chapter 2

Todo List

page API Specification (p. **??**) Move this documentation to another file or to the Sphinx documentation.

Global application_run (p. **9**)**()** Will this method ever return if the application starts correctly?

File manager.h (p. **7**) How can we read the name of a given application? Should we provide a function here or do we read the value directly from the configuration registry?

Global pstorage_erase (p. **19**)**(void ∗address, int length)** Define the possible error numbers and their respective meaning.

Global pstorage_read (p. **19**)**(char ∗buffer, void ∗address, int length)** Define the possible error numbers and their respective meaning.

Global pstorage_write (p. **20**)**(const char ∗buffer, void ∗address, int length)** Define the possible error numbers and their respective meaning.

File registry.h (p. **14**) Define the values in the `CONFIG_KEY` enum.

Should we make this interface more generic and not define the config keys using enum values? This woul allow for non hard coded keys but we loose compile-time error checking. If we change the interface in the described way, there will be the need for a `config_clear` function.

The two get/set functions are very generic and involve a relatively large amount of code to be used, wouldn't it be better to define the different types of the configuration directives (mainly strings and ints) and to offer type-specific getters and setters (i.e. `config_get_int`)?

Global serial_readchar (p. **16**)**()** How can we handle errors of this function? Setting a timeout limit and a special return code could do the trick.

Global serial_readline (p. **17**)**(char ∗buffer)** Should we set a timeout option? Should we allow to configure the char sequence to identify a line break (\n, \r\n or \r)?

Define the possible error numbers and their respective meaning.

Global vstorage_read (p. **21**)**(char ∗buffer, void ∗address, int length)** Define the possible error numbers and their respective meaning.

Global vstorage_write (p. **21**)**(const char ∗buffer, void ∗address, int length)** Define the possible error numbers and their respective meaning.

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

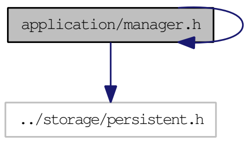
Chapter 4

File Documentation

4.1 application/manager.h File Reference

```
#include "../storage/persistent.h"
#include "../storage/persistent.h"
```

Include dependency graph for manager.h:



This graph shows which files directly or indirectly include this file:

Enumerations

- enum **APPLICATION_ID** { **APP_0**, **APP_1** }

Functions

- int **application_load** (enum **APPLICATION_ID** appid)
- int **application_run** ()
- int **application_add** (enum **APPLICATION_ID** appid, const char ∗name, const char ∗buffer)
- int **application_remove** (enum **APPLICATION_ID** appid)
- int **application_rename** (enum **APPLICATION_ID** appid, const char ∗name)

4.1.1 Detailed Description

Application management utilitites. This functions allow to run, add and manage existing applications.

Todo

How can we read the name of a given application? Should we provide a function here or do we read the value directly from the configuration registry?

Definition in file **manager.h**.

4.1.2 Enumeration Type Documentation

4.1.2.1 enum APPLICATION_ID

The different IDs given to the application which can possibly be hosted on the target system. The number of values in the enumeration gives the maximum number of application which can coexist on the target.

Enumerator:

APP_0

APP_1

Definition at line 25 of file manager.h.

4.1.3 Function Documentation

4.1.3.1 int application_add (enum APPLICATION_ID *appid*, const char ∗ *name*, const char ∗ *buffer*)

Adds the application at *buffer* to the slot identified by *appid*, giving it the name *name*.

Parameters

← ***appid*** The slot to which write the application to the persistent storage.

← ***name*** The human readable name to give to this application.

← ***buffer*** The buffer from which to read the application to be written.

Returns

0 if the operation completed successfully, an error code otherwise.

4.1.3.2 int application_load (enum APPLICATION_ID *appid*)

Loads the application identified by *appid* to the RAM.

Parameters

← ***appid*** The identificator for the application to load.

Returns

0 if the application was loaded correctly, an error code if the application failed to load.

4.1.3.3 int application_remove (enum APPLICATION_ID *appid*)

Removes the application currently located at the slot *appid* from the system.

Parameters

← ***appid*** The slot in the persistent storage to free up.

Returns

0 if the operation completed successfully, an error code otherwise.

4.1.3.4 int application_rename (enum APPLICATION_ID *appid*, const char ∗ *name*)

Renames the application currently located at the slot *appid*, giving it the name *name*.

Parameters

← ***appid*** The slot in the persistent storage where the application to rename is located.

← ***name*** The new name to assign to this application.

4.1.3.5 int application_run ()

Runs the application currently stored in the RAM.

Todo

Will this method ever return if the application starts correctly?

Returns

An error code indicating the reason of the failure.

4.2 commands/dispatcher.h File Reference

`#include "registry.h"`

Include dependency graph for dispatcher.h:

This graph shows which files directly or indirectly include this file:

Defines

- #define **MAX_ARGC** 100

Functions

- int **command_parse** (const char ∗cmd, char ∗args[ ])
- int **command_dispatch** (int argc, char ∗argv[ ])
- int **command_parse_and_dispatch** (const char ∗cmd)

4.2.1 Detailed Description

This files contains various utilities which operate on the commands registry to parse a shell like command string and to call the command with the retrieved arguments.

Definition in file **dispatcher.h**.

4.2.2 Define Documentation

4.2.2.1 #define MAX_ARGC 100

The maximum number of tokens which an array should be able to contain.

Definition at line 17 of file dispatcher.h.

4.2.3 Function Documentation

4.2.3.1 int command_dispatch (int *argc*, char ∗ *argv*[ ])

Takes a list of tokens and its count and executes the registered command. This function takes the first ite in *argv* and retrieves the command from the registry using this string.

Parameters

← ***argc*** The number of tokens contained in *argv*.

$\leftarrow$ ***argv*** The tokens as returned by the `command_parse` function.

Returns

The status code of the executed command (either 0 or a positive error code if the command execution failed) or a negative error code if it was not possible to call the command.

4.2.3.2 int command_parse (const char $*$ *cmd*, char $*$ *args*[ ])

Parses a command string and populates an array with the obtained tokens.

Precondition

args is an array of size `MAX_ARGC`

Parameters

$\leftarrow$ ***cmd*** The string containing the command to parse.

$\rightarrow$ ***args*** The array which will be populated with the tokens obtained by parsing the *cmd* argument.

Returns

The number of found tokens (the count of items in *args*) or a negative error code if the function fails.

4.2.3.3 int command_parse_and_dispatch (const char $*$ *cmd*)

Shortcut function to directly parse and dispatch a command in one step. This function calls `command_-parse` and passes its results to `command_dispatch`

Parameters

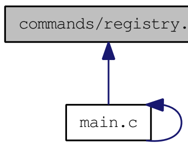
$\leftarrow$ ***cmd*** The string containing the command to parse.

Returns

The value returned by the `command_dispatch` function call.

4.3 commands/registry.h File Reference

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef int(∗ **Command**)(int argc, const char ∗argv[ ])

Functions

- int **command_registry_register** (const char ∗name, **Command** cmd)
- int **command_registry_unregister** (const char ∗name)
- **Command command_registry_get** (const char ∗name)

4.3.1 Detailed Description

A registry which allows to add, remove and get pointers to Command like functions using a given name as identifier. This data structure is mainly used by the dispatcher to store all the known commands and retrieve them by name.

Definition in file **registry.h**.

4.3.2 Typedef Documentation

4.3.2.1 typedef int(∗ Command)(int argc, const char ∗argv[ ])

The interface which all functions used as commands have to implement. The interface is similar to the required prototype for the `main` method; it takes an *argc* argument indicating the number of arguments and an `argv` argument which points to an array of strings, already splitted by the command parser. The function must return 0 if completed successfully or a custom positive error code if it fails.

Definition at line 22 of file registry.h.

4.3.3 Function Documentation

4.3.3.1 Command command_registry_get (const char ∗ *name*)

Retrieves and returns the command identified by the given name.

Parameters

← ***name*** The string indicating the name of the command to retrieve.

Returns

The command if a value existed for the key name, NULL otherwise.

4.3.3.2 int command_registry_register (const char $*$ *name*, Command *cmd*)

Adds a given command to the commands registry using the given name. If a command was already registered with the same name, it is overwritten.

Parameters

$\leftarrow$ ***name*** The string indicating the name to use to identify the command.

$\leftarrow$ ***cmd*** The pointer to the command function to register under the given name.

Returns

0 if the value was read correctly, a negative error code otherwise.

4.3.3.3 int command_registry_unregister (const char $*$ *name*)

Removes the command identified by the given name from the command registry.

Parameters

$\leftarrow$ ***name*** The string indicating the name of the command to remove.

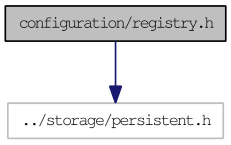
Returns

0 if the command was removed correctly, 1 if it was not found and a negative error code if an error occurred.

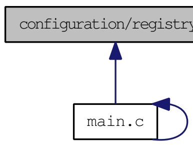
4.4 configuration/registry.h File Reference

`#include "../storage/persistent.h"`

Include dependency graph for registry.h:



This graph shows which files directly or indirectly include this file:



Enumerations

- enum **CONFIG_KEY** { **DUMMY_CONFIG_KEY** }

Functions

- int **config_get** (enum **CONFIG_KEY** key, void ∗buffer)
- int **config_set** (enum **CONFIG_KEY** key, const void ∗buffer)

4.4.1 Detailed Description

This module allows to store to and retrieve configuration values from a persistent storage device. It uses the **storage/persistent.h** (p. 19) interface for all read/write operations.

Todo

Define the values in the `CONFIG_KEY` enum.
Should we make this interface more generic and not define the config keys using enum values? This woul allow for non hard coded keys but we loose compile-time error checking. If we change the interface in the described way, there will be the need for a `config_clear` function.
The two get/set functions are very generic and involve a relatively large amount of code to be used, wouldn't it be better to define the different types of the configuration directives (mainly strings and ints) and to offer type-specific getters and setters (i.e. `config_get_int`)?

Definition in file **registry.h**.

4.4.2 Enumeration Type Documentation

4.4.2.1 enum CONFIG_KEY

Defines the different available configuration directives. This keys can be used to get or set configuration values with the `config_get` and `config_set` functions.

Enumerator:

DUMMY_CONFIG_KEY

Definition at line 31 of file registry.h.

4.4.3 Function Documentation

4.4.3.1 int config_get (enum CONFIG_KEY *key*, void ∗ *buffer*)

Reads the value identified by *key* argument from the config registry and writes it to the memory area pointed to by *buffer*.

Parameters

$\leftarrow$ ***key*** The key of the configuration directive to read.

$\rightarrow$ ***buffer*** The pointer to the memory area to which the value has to be written.

Returns

0 if the value was read correctly, a negative error code otherwise.

4.4.3.2 int config_set (enum CONFIG_KEY *key*, const void ∗ *buffer*)

Sets the value identified by *key* to the content of *buffer*.

Parameters

$\leftarrow$ ***key*** The key of the configuration directive to read.

$\leftarrow$ ***buffer*** The pointer to the memory area from which the value has to be read.

Returns

0 if the value was set correctly, a negative error code otherwise.

4.5 io/serial.h File Reference

Functions

- int **serial_init** (const void ∗config_struct)
- char **serial_readchar** ()
- int **serial_readline** (char ∗buffer)
- int **serial_write** (const char ∗buffer)

4.5.1 Detailed Description

Interface to a serial interface (wow, an interface of an interface... :-)

The prototypes contained in this file can be implemented multiple times to be adapted to different communication interfaces which can simulate a serial port (e.g. a real serial port, a TCP socket, a Unix socket,...).

Definition in file **serial.h**.

4.5.2 Function Documentation

4.5.2.1 int serial_init (const void ∗ *config_struct*)

Initializes the serial port using the values contained in the given struct. The type of the struct is not defined to allow each implementation to use a struct containing the values needed to set up its own communication interface.

For example, an implementation which connects to a serial port uses a struct which contains values to define the speed, partity and stop bits of the interface, while a testing implementation which simulates a serial port using a TCP socket needs to know the values for the host and port to which establish the connection.

Parameters

← ***config_struct*** An implementation dependent struct to hold the needed configuration settings.

Returns

0 if the initialization process ended sucessfully, an error code otherwise.

4.5.2.2 char serial_readchar ()

Reads a single char from a serial interface, blocking until some character data is received.

Precondition

The `serial_init` function was already called and returned sucessfully.

Todo

How can we handle errors of this function? Setting a timeout limit and a special return code could do the trick.

Returns

The char read from the communication device.

4.5.2.3 int serial_readline (char $*$ *buffer*)

Reads data from the communication device until a newline character is encountered, then saves the data to the buffer and returns the number of bytes read.

Precondition

The `serial_init` function was already called and returned sucessfully.

Postcondition

The data saved to the buffer does not contain any newline characters and is \0 terminated.

Todo

Should we set a timeout option? Should we allow to configure the char sequence to identify a line break (\n, \r\n or \r)?
Define the possible error numbers and their respective meaning.

Parameters

$\rightarrow$ ***buffer*** The pointer to the string buffer to which the read line has to be written.

Returns

The number of characters written to the buffer if the operation completed successfully, a negative error code otherwise.

4.5.2.4 int serial_write (const char $*$ *buffer*)

Writes the data contained in the given character buffer to the communication device until the end of the string (\0).

Precondition

The `serial_init` function was already called and returned sucessfully.
The *buffer* string is correctly \0 terminated.

Parameters

$\leftarrow$ ***buffer*** The string to write to the serial interface.

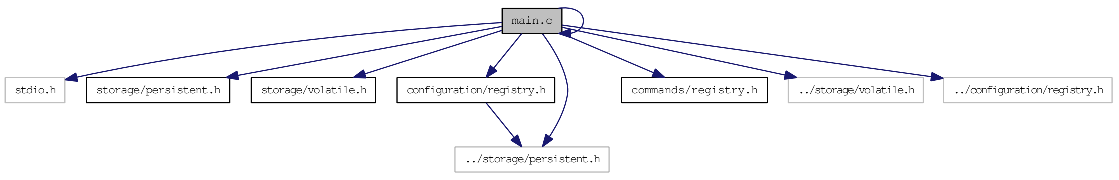
Returns

The number of characters written if the operation completed successfully, a negative error code otherwise.

4.6 main.c File Reference

```
#include <stdio.h>
#include "storage/persistent.h"
#include "storage/volatile.h"
#include "io/serial.h"
#include "configuration/registry.h"
#include "commands/registry.h"
#include "../storage/persistent.h"
#include "../storage/volatile.h"
#include "../configuration/registry.h"
```

Include dependency graph for main.c:



This graph shows which files directly or indirectly include this file:

Functions

- int **main** (int argc, const char ∗argv[ ])

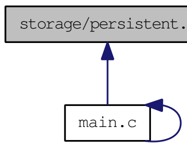
4.6.1 Function Documentation

4.6.1.1 int main (int *argc*, const char ∗ *argv*[ ])

Definition at line 60 of file main.c.

4.7 storage/persistent.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- int **pstorage_read** (char ∗buffer, void ∗address, int length)
- int **pstorage_write** (const char ∗buffer, void ∗address, int length)
- int **pstorage_erase** (void ∗address, int length)

4.7.1 Detailed Description

Interface to read/write/erase operations on a persistent storage device.

The prototypes contained in this file can be implemented multiple times to be adapted to different storage devices such as Hard Disk Drives, Flash memory devices or RAMs (i.e. for development and testing purposes).

Definition in file **persistent.h**.

4.7.2 Function Documentation

4.7.2.1 int pstorage_erase (void ∗ *address*, int *length*)

Deletes some data from the permanent storage handler compiled with this function.

Todo

Define the possible error numbers and their respective meaning.

Parameters

← ***address*** The address from which to start to delete the data from the storage device.

← ***length*** The number of bytes to delete from the storage device, starting at the given address.

Returns

The number of bytes deleted, a negative number if an error occurred.

4.7.2.2 int pstorage_read (char ∗ *buffer*, void ∗ *address*, int *length*)

Reads some data from the permanent storage handler compiled with this function.

Todo

Define the possible error numbers and their respective meaning.

Parameters

$\rightarrow$ ***buffer*** The pointer to a char buffer where the read data has to be written.

$\leftarrow$ ***address*** The address from which to start to read from the storage device.

$\leftarrow$ ***length*** The number of bytes to read from the storage device, starting at the given address.

Returns

The number of bytes read, a negative number if an error occurred.

4.7.2.3 int pstorage_write (const char $*$ *buffer*, void $*$ *address*, int *length*)

Writes some data to the permanent storage handler compiled with this function.

Todo

Define the possible error numbers and their respective meaning.

Parameters

$\rightarrow$ ***buffer*** The pointer to a char buffer from where the data to write has to be read.

$\leftarrow$ ***address*** The address from which to start to write to the storage device.

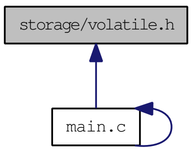
$\leftarrow$ ***length*** The number of bytes to write to the storage device, starting at the given address.

Returns

The number of bytes written, a negative number if an error occurred.

4.8 storage/volatile.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- int **vstorage_read** (char ∗buffer, void ∗address, int length)
- int **vstorage_write** (const char ∗buffer, void ∗address, int length)

4.8.1 Detailed Description

Interface to read/write operations on a volatile storage devices or interfaces.

The prototypes contained in this file can be implemented multiple times to be adapted to different storage devices such as RAMs, Cache chips or Hard Disk Drives (i.e. for development/testing purposes).

Definition in file **volatile.h**.

4.8.2 Function Documentation

4.8.2.1 int vstorage_read (char ∗ *buffer*, void ∗ *address*, int *length*)

Reads some data from the volatile storage handler compiled with this function.

Todo

Define the possible error numbers and their respective meaning.

Parameters

$\rightarrow$ ***buffer*** The pointer to a char buffer where the read data has to be written.

$\leftarrow$ ***address*** The address from which to start to read from the storage device.

$\leftarrow$ ***length*** The number of bytes to read from the storage device, starting at the given address.

Returns

The number of bytes read, a negative number if an error occurred.

4.8.2.2 int vstorage_write (const char ∗ *buffer*, void ∗ *address*, int *length*)

Writes some data to the volatile storage handler compiled with this function.

Todo

Define the possible error numbers and their respective meaning.

Parameters

$\rightarrow$ ***buffer*** The pointer to a char buffer from where the data to write has to be read.

$\leftarrow$ ***address*** The address from which to start to write to the storage device.

$\leftarrow$ ***length*** The number of bytes to write to the storage device, starting at the given address.

Returns

The number of bytes written, a negative number if an error occurred.

Index