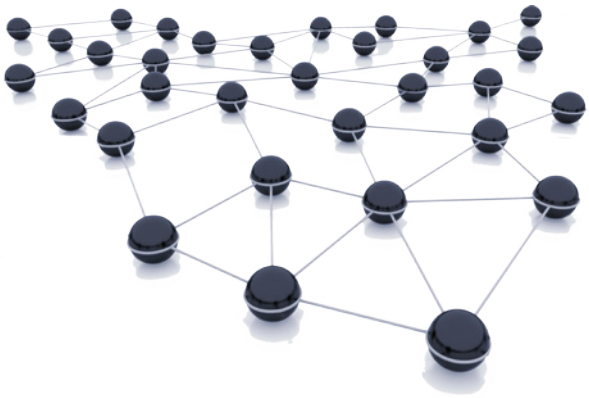
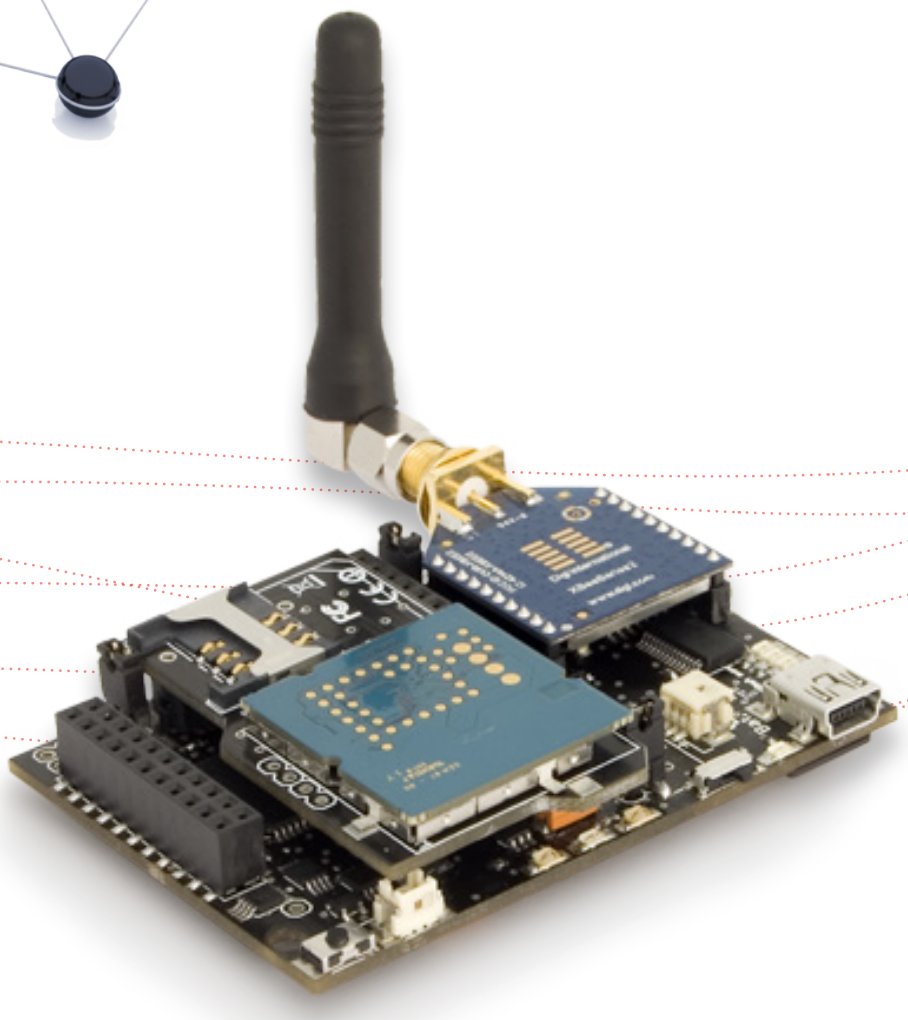
Waspmote SD CARD

Programming Guide





libelium

waspmote

Document Version: v0.9 - 01/2014

INDEX

Important Note:

This Guide is only intended for Waspmote v11, the first Waspmote version. Although Waspmote v11 was discontinued in February 2013 with the release of the new version (Waspmote v12), Libelium wanted to collect all the benefits present in Waspmote v12 library and port them all to Waspmote v11, since there are still many users depeloping with this old Waspmote.

So this Guide applies to the new Waspmote v11 API, called API v0.33 (December 2013). This API delivers deep changes in WiFi, GPRS and SD libraries.

Most of the features of the Waspmote v12 API v005 library were ported to the new library for Waspmote v11. Most of the bugs were fixed too. Anyhow, you may expect some features were not ported or some bugs were not fixed. For the best experience with Waspmote, give a try to Waspmote v12.

Note: *There are many SD card models. Any of them has defective blocks, which are ignored when using the Waspmote's SD library. However, when using OTA, those SD blocks cannot be avoided, so that the execution could crash. Libelium implements a special process to ensure the SD cards we provide will work fine with OTA. The only SD cards that Libelium can assure that work correctly with Waspmote are the SD cards we distribute officially.*

Note: *Make sure Waspmote is switched off before inserting or removing the SD card. Otherwise, the SD card could be damaged.*

Note: *Waspmote must not be switched off or reseted while there are ongoing read or write operations in the SD card. Otherwise, the SD card could be damaged and data could be lost. If you suspect that there may be some ongoing SD operations, wait a while until they are completed.*

Tip: *you can use one programmable LED to signal when the SD card is being processed.*

1. General Considerations

1.1. Waspmote Libraries

1.1.1. Waspmote SD Files

WaspSD.h ; WaspSD.cpp

Other utilities: Sd2Card.cpp; Sd2Card.h; Sd2Fat.h; Sd2FatStructs.h; Sd2File.cpp; Sd2Info.h; Sd2PinMap.h; Sd2Volume.cpp

1.1.2. Constructor

To start using Waspmote SD library, an object from class 'WaspSD' must be created. This object, called 'SD', is created inside Waspmote SD library and it is public to all libraries. It is used through the guide to show how Waspmote SD library works.

When creating this constructor, no variables are initialized by default.

1.1.3. Flag

A flag to indicate if there have been any problem during execution of a function has been created. This flag shows the state of the SD card during initialization and operation. Possible values are:

- 0 : nothing failed, all processes have been executed properly.
- 1 : no SD card in the slot.
- 2 : initialization failed.
- 4 : volume partition failed.
- 8 : root failed.
- 16 : truncated data. Data length is bigger than buffer length, so data will be truncated to fit the buffer.
- 32 : error when opening a file.
- 64 : error when creating a file.
- 128 : error when creating a directory.
- 256 : error when writing to a file.

1.1.4. Error Messages

When executing some functions, a string is returned explaining the state of SD card. Possible messages are:

- "no SD" : SD has not been found in the card slot
- "Invalid filename": An invalid name for a file or a directory. It must respect the "8.3 filename" format (also called "short filename" or "SFN")

1.1.5. Buffer

Due to memory restrictions, a buffer has been created to limit the length of the data managed by Waspmote API libraries when working with SD cards. Buffer size has been set to 256Bytes due to it is a sufficient value to manage strings and it occupies little memory.

This limit must be considered when developing applications, using the flag previously explained to know when data has been truncated. Obviously this does not mean you can handle 256B only, but you have to make writings and readings of this size.

1.2. Formatting SD Card

When formatting the SD card before starting using Waspmote, there are some considerations to have in mind. The most important matter to know before formatting an SD card is setting the right size of the allocation tables to address the card properly. Despite of selecting FAT16, when formatting the card if there is no indication, the OS will select a size between 12,16 and 32 bits. The right value is 16b.

1.3. Short filename format

An 8.3 filename (also called a short filename or SFN) is a filename convention which is followed by the SD card library. 8.3 filenames have at most eight characters, optionally followed by a "." character and a filename extension of at most three characters.

Only upper-case letters A–Z are valid. When using lower-case letters a-z, these are converted to upper-case letter.

Besides, illegal characters for directories and filenames include the following:

| < > ^ + = ? / [ ] ; , * \ " \ \

Example of valid and invalid filenames:

FILE.TXT → valid

FILENAME → valid

FILENAME.TXT → valid

FOLDER → valid

SUBFOLDER → invalid (more than 8 characters)

FILENAME1.TXT → invalid (more than 8 characters before ".")

FILENAME.AAAA → invalid (more than 3 characters after ".")

file.txt → valid (but it will be interpreted as FILE.TXT)

2. Initialization

Before start using the SD card, it needs to be initialized. This process checks if an SD card is present on the slot, initializes SPI bus, opens partition and opens root directory.

2.1. Initializing a Card

The following function checks if an SD card is present in the slot, sets the microcontroller pin which powers the SD card up, initializes the SPI bus, opens the FAT volume partition and opens the root directory. It returns nothing but it updates the flag with an error code indicating the possible error messages.

Example of use

```
{
   SD.ON(); // Set SD card on
}
```

Available Information

SD.flag → stores the error code indicating the state of SD initialization process.

The SD card cannot be removed or inserted without powering off Waspmote. If a SD card is removed, the initialization function should be called to initialize the card again. Before that, the SD Card should be closed using functions explained in section "Closing a card".

2.2. Closing a Card

Closes the root directory and the SPI bus. It also switches off the microcontroller pin that powers the SD card. It returns nothing and it does not change the flag value.

Example of use

```
{
   SD.OFF(); // Powers SD card down
}
```

2.3. SD Present

It reads the associated pin to know if there is an SD in card slot.

It returns '1' if SD card is present and '0' if not.

If SD is not present, it closes card to avoid problems with pointers.

Example of use

```
{
        uint8_t present;

        // Reads associated pin to know if there is a SD in card slot
        present=SD.isSD();
}
```

Available Information

present→ stores '1' if SD card is detected and '0' if not.

3. Disk Operations

When SD has been initialized properly, pointers can be used to access partition, file system and root directory. There are some functions that return information about the SD card.

3.1. Disk Information

Stores all the data containing the disk info into the buffer. It returns a filled buffer if success on getting disk info and an empty string if not.
Example of use

```
{
  // Gets disk info, returning it and storing this info in 'SD.buffer'
  SD.print_disk_info();

  USB.println(SD.buffer);
}
```

Available Information

`SD.buffer` → stores the data received as a human-readable encoded string.
`diskInfo` → pointer to `SD.buffer`

An example of the output by this system would be:

```
manuf:  0x1b
oem:    SM
prod:   21e7
rev:    1.0
serial: 0xedb6c604
date:   11/10
```

3.2. Disk Size

Gets the total size of the SD card.

It returns 'diskSize' variable and updates its value. Size is stored and returned in Bytes.

Example of use

```
{
  // Get total size of SD card
  diskSize = SD.getDiskSize();

  USB.println(SD.diskSize);
}
```

Related Variables

SD.diskSize → stores the total size of the SD card.

An example of the value would be: SD.diskSize=992608256

4. Directory Operations

To organize an SD card, it is possible to create and manage directories. There are some functions related with directories.

4.1. Creating a Directory

It creates a directory given as a valid directory path (according to short filename format) in the current working directory. The root directory is the default directory each time SD card is initialized.

It returns '1' on creation and '0' on error, activating the flag too.

If a directory name already exists, it will occur an error and the flag will be activated.

Example of use

```
{
   boolean dirCreation;
   char* name = "FOLDER1";
   char* path = "FOLDER3/FOLDER4/FOLDER5";

   // creates a directory in the current directory called "FOLDER1"
   dirCreation = SD.mkdir(name);

   // creates a directory in the current directory called "FOLDER2"
   dirCreation = SD.mkdir("FOLDER2");

   // creates a three-directory path in the current directory
   dirCreation = SD.mkdir(path);
}
```

Note 1: *All directory names must be defined according to 8.3 short filename format (see section "Short filename format")*

Note 2: *Be careful when calling this function to create a directory. If it is interrupted, the directory results damaged and it is necessary to delete it as a regular file using* `SD.del`

4.2. Deleting a Directory

Empty directories

The directory file will be removed only if it is empty and is not the root directory.

It returns '1' if the directory has been erased properly and '0' if error.

Note: *It allows erasing a complete path of directories always they are empty.*

Example of use

```
{
   const char* name="FOLDER";
   char* path="FOLDER3/FOLDER4/FOLDER5";
   uint8_t delState;

   // deletes the directory in the current directory called "FOLDER"
   delState = SD.rmdir(name);

   //deletes the directory in the current directory called "FOLDER2"
   delState = SD.rmdir ("FOLDER2");

   //deletes a three-empty-directory path in the current directory
```

```
   dirCreation = SD.rmdir(path);
}
```

Non-empty directories

It is possible to delete a directory and all contained files. It returns '1' if the directory has been erased properly and '0' if error.

Example of use

```
{
   const char* name = "FOLDER";
   char* path = "FOLDER3/FOLDER4/FOLDER5";
   boolean delState;

   // deletes the directory in the current directory called "FOLDER"
   delState = SD.rmRfDir(name);

   // deletes the directory in the current directory called "FOLDER2"
   delState = SD.rmRfDir ("FOLDER2");

   // deletes a three-directory path in the current directory
   dirCreation = SD.rmRfDir ("FOLDER3");
}
```

4.3. Directory Listing

Prints through the USB port the contents of the current working directory. It is possible to introduce three different flags which may be an inclusive OR of:

LS_DATE - Print file modification date

LS_SIZE - Print file size.

LS_R - Recursive list of subdirectories.

It returns nothing. The information is printed through the USB port.

Example of use

```
{
   // lists the name of all the files of the directory indicating the size of the files
   SD.ls();

   // lists the name of all files of the directory indicating the size of the files
   SD.ls(LS_SIZE);

   // lists the name of all files of the directory indicating the date of the files
   SD.ls(LS_DATE);

   // lists the name of all files of the directory and all subdirectories
   SD.ls(LS_R);

   //lists the name of the files recursively indicating size and date
   SD.ls(LS_R|LS_DATE|LS_SIZE);
}
```

An example of the output by `SD.ls(LS_R|LS_DATE|LS_SIZE);` would be:

```
FILE8      1980-01-01 00:00:00 204
```

```
FILE2       1980-01-01 00:00:00 2754
FOLDER/     2000-01-01 01:00:00
  SUBFOLD/    2000-01-01 01:00:00
FILE.TXT    2012-06-11 11:58:10 811
```

4.4. Finding a Directory

It finds a sub-directory in the current directory. If it exists and it is a directory '1' will be returned, '0' will be returned if it exists but it is not a directory and '-1' will be returned if it does not exist.

Example of use

```
{
   uint8_t isdir;
   const char* name = "FOLDER";

   // tests existence of "FOLDER" in the current directory
   isdir = SD.isDir(name);

   // tests existence of "FOLDER" in the current directory
   isdir = SD.isDir("FOLDER");
}
```

4.5. Number of files

It gets the amount of files and subdirectories in the current directory. It returns the number of files or directories found, or zero if there are no files or directories. It does not count '.' and '..' directories, so if there are no directories or files in the current directory, zero will be returned.

If an error occurs, a negative number is returned.

Example of use

```
{
   int8_t numfiles;

   // returns the number of files in the current directory
   numfiles = SD.numFiles();
}
```

4.6. Changing Directory

It changes the current working directory pointer to the directory given as a parameter. It returns '0' if error, and '1' if not.

Note: *In root directory it has no sense changing directory to '..', so function will return error when doing that.*

Example of use

```
{
   uint8_t cdState;
   const char* command = "FOLDER";

   // Change to directory specified in 'command'
   cdState = SD.cd(command);

   // Go one directory up
```

```
   cdState = SD.cd("..");
}
```

Go directly to root directory

It is possible to go to root directory using a simple called 'goRoot'. It returns '1' when ok, '0' when error.

Example of use

```
{
   uint8_t cdState;

   // define the directory path to change
   char* path = "/FOLD1/FOLD2/FOLD3/FOLD4/FOLD5";

   // Change to 'fold5' directory specified in 'path'
   cdState = SD.cd(path);

   // Go to root directory
   cdState = SD.goRoot();
}
```

5. File Operations

To store data, files can be created and managed. There are some functions related to files operations.

5.1. Creating Files

It creates a file.

It returns '1' on file creation and '0' if error and it will mark the flag too.

Example of use

```
{
   const char* name = "FILE.TXT";
   boolean fileCreation;

   // It creates a file named "FILE.TXT"
   fileCreation = SD.create(name);

   // It creates a file named "FILE.TXT"
   fileCreation = SD.create("FILE.TXT");
}
```

Note 1:*All file names must be defined according to 8.3 short filename format (see section "Short filename format")*

Note 2: *The maximum number of files which can be created in the root directory are 341, while a 2nd level directory can store 1.000+ files. Thus, in the case the user needs to create hundreds of files in an SD card, it is highly advised to create them inside a 2nd level directory.*

5.2. Deleting Files

It deletes a file in the current directory. It returns '1' on file delete and '0' if error.

Example of use

```
{
   const char* name = "FILE.TXT";
   boolean fileDelete;

   // It deletes a file named "FILE.TXT"
   fileDelete = SD.del(name);

   // It deletes previously created file named "FILE.TXT"
  fileDelete = SD.del("FILE.TXT");
}
```

5.3. Opening Files

Opens the filepath if available. It is possible to select a bitwise-inclusive OR of flags from the following list:

O_READ - Open for reading.
O_WRITE - Open for writing.
O_RDWR - Open for reading and writing.
O_APPEND - If set, the file offset shall be set to the end of the file prior to each write.
O_CREAT - If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file shall be created.
O_EXCL - If O_CREAT and O_EXCL are set, open() shall fail if the file exists.
O_SYNC - Synchronous writes.

Returns '1' on success, '0' otherwise

Example of use

```
{
   const char* filepath = "FILE.TXT";

   // declare an SdFile object
   SdFile file;

   // open "FILE" for reading
   SD.openFile( filepath, &file, O_READ);
}
```

Note: *All file names must be defined according to 8.3 short filename format (see section "Short filename format")*

5.4. Closing Files

It closes the pointer which pointed to the previously defined file.

Note: *If a file is opened with previous function and it is not closed before using another file function, it will not work properly. Only one file pointer can be managed at the same time.*

Example of use

```
{
   // previously declared file
   SdFile file;

   // close file, referencing the previously opened file
   SD.closeFile(&file);
}
```

5.5. Finding Files

It finds a file path in the current directory.

If it exists and it is a file '1' will be returned, '0' will be returned if it exists but it is not a file and '-1' will be returned if it does not exist.

Example of use

```
{
      const char* name = "FILE.TXT";
   int8_t fileFound;

   // looks for "FILE.TXT" in the current directory
   fileFound = SD.isFile(name);

   // looks for "FILE.TXT" in the current directory
   fileFound = SD.isFile("FILE.TXT");}
```

5.6. Reading data

It dumps into the buffer the amount of bytes/lines indicated in '`scope`' after '`offset`' bytes/lines.

The information is returned as a string where each one of the characters are printed one after the next, EOL ('\n') will be encoded as EOL, and will be accounted as one byte.

Note: *There is a limitation in size, due to buffer size. If the data read was bigger than that, the function will include the characters ">>" at the end and activate the* `TRUNCATED_DATA` *value in the flag. It is recommended to check this value to ensure data integrity.*

If 'offset' or 'scope', or both of them, are greater than file size, there will be no error but only possible data will be copied into buffer.

Example of use

```
{
   const char* name="FILE.TXT";

   //It stores in 'SD.buffer' 17 characters after jumping 3
   SD.cat(name,3,17);

   //It stores in 'SD.buffer' 100 characters from the beginning
   SD.cat(name,0,100);

   //It stores in 'SD.buffer' 3 lines after jumping over the 2 first
   SD.catln(name,2,3);
}
```

5.7. Writing Data

There are several forms of writing data to a file:

- Indicating the position to start writing. Function 'SD.writeSD'. It is possible to indicate the amount of bytes to write.
- At the end of the file. Function 'SD.append'. It is possible to indicate the amount of bytes to write.
- At the end of the file including an EOL character. Function 'SD.appendln'

It returns '1' on success and '0' if error.

Note: *Due to buffer size, 256Bytes is the limit for writing data into a file at once. If more data needs to be written, it will have to be divided in blocks of 256Bytes.*

Example of use

```
{
   const char* file = "FILE.TXT";
   uint8_t writeState;

   // It writes "hello" on file at position 0
   writeState = SD.writeSD(file,"hello",0);

   // It writes "hello" on file at position 5
   writeState = SD.writeSD(file,"hello",5);

   // It writes "hel" on file at position 10
   writeState = SD.writeSD(file,"hello",10,3);

   // It writes "hello" at the end of file
   writeState = SD.append(file,"hello");

   // It writes "hel" on file at end of file
   writeState = SD.append(file,"hello",3);

   // It writes "hello" at end of file with EOL
   writeState = SD.appendln(file,"hello");
}
```

5.8. Number of Lines

It counts the number of lines in a file.

The number of lines are counted as the number of '\n' that are found in a file.

It returns the amount of lines and negative value if an error occurred. If there are no lines in file selected, zero will be returned.

Example of use

```
{
   const char* name = "FILE.TXT";
   int32_t numberLines;

   // counts number of lines in file named "FILE.TXT"
   numberLines = SD.numln(name);

   // counts number of lines in file named "FILE.TXT"
   numberLines = SD.numln("FILE.TXT");
}
```

5.9. Getting File Size

It gets the size of the selected file.

It returns its size in Bytes or '-1' if an error occurs.

Example of use

```
{
   const char* name = "FILE.TXT";
   int32_t sizeFile;

   // gets size of file named "FILE.TXT"
   sizeFile = SD.getFileSize(name);

   // gets size of file named "FILE.TXT"
   sizeFile = SD.getFileSize("FILE.TXT");
}
```

Available Information

A possible value would be: `sizeFile=16`, indicating the size file is 16 Bytes.

5.10. Finding Patterns

It looks into the file for the first occurrence of the pattern after a certain offset. The algorithm will jump over offset bytes before starting to search for the pattern.

It will return the amount of bytes to the pattern from the offset.

The special characters like '\n' (EOL) are accounted as one byte and files are indexed from 0.

Example of use

```
{
    const char* name = "FILE.TXT";
   int32_t pattern;

   // It returns position at which "11" appears on file jumping over first 17 positions
   pattern = SD.indexOf(name,"11",17);

   // It returns position at which "11" is on file
   pattern = SD.indexOf("FILE.TXT","11",0);
}
```

Example file "FILE.TXT" contains: `'hola caracola\nhej hej\n hola la[EOF]'`

The following table shows the results from searching different patterns:

Command		Answer
SD.indexOf("FILE.TXT", "hola", 0)	→	0
SD.indexOf("FILE.TXT", "hola", 1)	→	23
SD.indexOf("FILE.TXT", "hej", 3)	→	11

6. Code examples and extended information

In the Waspmote Development section you can find complete examples:

http://www.libelium.com/development-v11/

7. API changelog

Function	Changelog	Version
FILE_SEEKING_ERROR	New definition	v0.29 → v0.33
WaspSD::cat	Internal changes in this function	v0.29 → v0.33
WaspSD::writeSD	Internal changes in this function	v0.29 → v0.33
WaspSD::numFiles	Bug fixed	v0.29 → v0.33
WaspSD::writeEndOfLine	New function to write EOF in file	v0.29 → v0.33
WaspSD::appendln	Changed all appendln functions to make sure to write the '\r' and '\n'	v0.29 → v0.33
WaspSD::ON	characters	v0.29 → v0.33
WaspSD::close	Changed	v0.29 → v0.33
WaspSD::getDiskFree();	Changed	v0.29 → v0.33
WaspSD::print_disk_info();	Different info is provided	v0.29 → v0.33
WaspSD::mkdir(const char* dirname);	Only SFN 8.3 (Short File Name) are permitted. It is possible to create complete paths of directories and subdirectories. i.e. /root/sub1/sub2/sub3. Prototype function has changed. Now this function returns boolean instead of uint8_t.	v0.29 → v0.33
WaspSD::ls(void);	This function now prints all the info related to files and directories. No "trash" files are printed. Prototype function has changed. Now this function returns void.	v0.29 → v0.33
WaspSD::ls(int offset);	It no longer exists	v0.29 → v0.33
WaspSD::ls(int offset, int scope, uint8_t info);	It no longer exists	v0.29 → v0.33
WaspSD::ls(uint8_t flags);	NEW FUNCTION. It is possible to list directories recursively adding info about the size and date.	v0.29 → v0.33
WaspSD::find_file_in_dir(const char* filepath);	Prototype of function changes. Now, there is only one input parameter.	v0.29 → v0.33
WaspSD::openFile(const char* filepath, SdFile* file, uint8_t mode);	Prototype of function changes. Now, file pointer is the first parameter. And open mode is the second parameter	v0.29 → v0.33
WaspSD::closeFile(SdFile* file);	Prototype of function changes. Now, file pointer is the parameter	v0.29 → v0.33
WaspSD::getAttributes(const char* name);	It no longer exists	v0.29 → v0.33
WaspSD::del(const char* filepath);	This function now only deletes files. It no longer deletes directories.	v0.29 → v0.33
WaspSD::rmDir(const char* dirname);	NEW FUNCTION. This function deletes empty directories.	v0.29 → v0.33
WaspSD::delFile(struct fat_dir_entry_struct file_entry);	It no longer exists	v0.29 → v0.33
WaspSD::delDir(uint8_t depth);	It no longer exists	v0.29 → v0.33
WaspSD::writeSD(const char* filename, const char* str, int32_t offset)	This function calls the same function but indicating the string length.	v0.29 → v0.33
WaspSD::writeSD(const char* filename, const char* str, int32_t offset, int16_t length)	This function has a new parameter in order to indicate the string length	v0.29 → v0.33

`WaspSD::writeSD( const char* filename, uint8_t* str, int32_t offset)`	This function calls the same function but indicating the calculated length with 0xAA 0xAA ending bytes.	v0.29 → v0.33
`WaspSD::writeSD( const char* filename, uint8_t* str, int32_t offset, int16_t length)`	NEW FUNCTION. Array length is indicated as a new parameter	v0.29 → v0.33
`WaspSD::append( const char* filename, uint8_t* str)`	NEW FUNCTION. Array of bytes is written at the end of the file	v0.29 → v0.33
`WaspSD::goRoot()`	NEW FUNCTION. Root directory is set as current working directory	v0.29 → v0.33
`WaspSD::rmRfDir(const char* dirpath);`	NEW FUNCTION. This new function deletes the selected directory and all contained files and subdirectories	v0.29 → v0.33

8. Documentation changelog

From v0.8 to v0.9

- API changelog created for the deep changes in libraries from API v0.32 to API v0.33