

# KokkeKat FAT-free SD card library

Dear community

This application note presents my new SD card library. It is written entirely in BASCOM AVR basic with the intention to minimize RAM usage.

While there might be license fees to pay to the SD Association ([www.sdcard.org](http://www.sdcard.org)) and Microsoft ([www.microsoft.com](http://www.microsoft.com)), this library itself doesn't have a license fee. I publish it for the benefit of the BASCOM community because I myself have received a lot of valuable help from other members.

License terms:

- You may use it in commercial and non-commercial BASCOM AVR applications without a license fee to me.
- You may not sell or redistribute this code or any part thereof in any way without a formal written agreement with me. This also applies to anybody selling or redistributing software you build upon this library and so on.
- You are responsible for any and all license fees to the SD Association, Microsoft, and other parties.
- This library is provided completely "as is". It comes without any guarantee or responsibility from my part whatsoever. Use it at your own risk.
- Without obligation and as far as I can find the time, I will try to provide support and bug fixes in a dedicated thread in the user forum at [www.mcselec.com](http://www.mcselec.com).

I hope you will find it useful. Please report back any bugs you encounter.

Niclas Arndt

Stockholm, Sweden, 2011.05.06

## 1 What makes it special?

- Written in commented basic so you can easily understand it and make any addition or modification you see fit.
- Low dedicated RAM requirement – read, write, and append with fsinfo updating takes only 651 bytes of RAM plus 10 bytes of \$hwstack . Read-only requires only 588 bytes of RAM plus 10 bytes of \$hwstack.
- An integrated directory scrolling routine useful for forward and backward navigation applications, e.g. mp3 players.
- Support for long filenames when scrolling and reading files.

## 2 Main features

- Modular – you can easily define which parts of the library you want to use.
- SDSC, SDHC, and SDXC support. (For SDXC you need to format it with FAT32 in Mac OS or Linux.)
- FAT16 and FAT32 support in root directory and subdirectory (no support for ExFAT).
- This means that the maximum file size is 4 GB and the maximum partition size is 2 TB. It has been development tested on 32 MB SDSC, 1 GB SDSC, 4 GB SDHC, and 64 GB SDXC.
- Find file, subdirectory, parent directory, and volume ID.
- Read file with 8.3 or long filename.
- Create and write to new file with 8.3 filename.
- Create new subdirectory with 8.3 directory name.
- Append to a pre-existing file with 8.3 filename.
- Scroll forward or backward through a directory.
- Update FAT32 fsinfo sector when writing.
- Wipe out the contents while maintaining the original formatting.

## 3 Requirements

- You must use commercial v2.0.5.0 (or higher) of BASCOM AVR. This is because this library requires the new data type DWORD: unsigned 32-bit integer.

## 4 Table of Contents

1	What makes it special? .....	2
2	Main features .....	2
3	Requirements .....	2
5	What you need to know before you start using the library .....	5
5.1	SD card versions and modes .....	5
5.2	SD card initialization and identification .....	5
5.3	SD card signalling speed.....	5
5.4	Block (sector) size .....	5
5.5	Understanding FAT16 and FAT32 .....	6
5.6	Byte in sector, sector in cluster, cluster, and absolute sector .....	6
5.7	Root directory .....	7
5.8	FAT, File Allocation Table .....	7
5.9	Fsinfo .....	8
5.10	Long filenames .....	8
6	Using the library .....	9
6.1	Configuration .....	9
6.2	Knowing your whereabouts .....	9
6.3	The root directory .....	9
6.4	Creating new files and directories and appending to files .....	10
6.4.1	Reading a file .....	10
6.4.2	Creating a new file .....	10
6.4.3	Creating a new subdirectory.....	10
6.4.4	Appending to a pre-existing file .....	11
6.5	Write safety vs. performance .....	11
6.6	Copying a file or reading and writing concurrently .....	11
6.7	Wiping the SD card.....	11
6.8	Examples .....	12
6.8.1	Initialization sequence.....	12
6.8.2	Code size and RAM usage for some of the examples.....	12
6.9	Sdstatus .....	12
7	General information about BASCOM AVR's use of RAM .....	13
7.1	Hardware Stack .....	13

7.2	Software Stack .....	13
7.3	Frame .....	14
7.4	RAM organization .....	15

## 5 What you need to know before you start using the library

### 5.1 SD card versions and modes

This library is based on Physical Layer Simplified Specification Version 3.01 (but according to the specs, SPI mode is the same as in version 2.00). In April 2011, this is the latest version.

SD cards have two basic modes: 4-bit and 1-bit Data Transfer mode and simplified 1-bit SPI mode. This library (and most other micro controller libraries) supports SPI mode only.

Version 1.10 defines SD Standard Capacity cards up to and including 2 GB.

Version 2.00 adds SD High Capacity and SD eXtended Capacity cards and an augmented initialization sequence. This means that there are currently at least four types of cards:

- Version 1.0 SDSC card (with short initialization sequence and Default Speed signalling)
- Version 1.X SDSC card (with short initialization sequence and optional High Speed signalling)
- Version 2.00 or later SDSC card with long initialization sequence
- Version 2.00 or later SDHC or SDXC card with long initialization sequence

This library supports all four card types.

Version 3.01 adds functionality not supported in SPI mode.

### 5.2 SD card initialization and identification

The initialization sequence sets the card in SPI mode, determines card type, and checks the card's supported voltage range. Upon completion, the card is ready for the actual reading and writing you are about to do.

### 5.3 SD card signalling speed

During initialization, communication with the card must be in the range of 100 kHz to 400 kHz. Depending on your AVR clock frequency, you might have to adjust the HW SPI declaration in the library.

After initialization, the maximum 3.3V (2.7 to 3.6V) signalling speed depends on the card's capability; Default Speed is  $\leq 25$  MHz and High Speed is  $\leq 50$  MHz.

With HW SPI, the maximum signalling speed is half the clock frequency ( $20 \text{ MHz} * 0,5$  for Mega or  $32 \text{ MHz} * 0,5$  for XMega), so the initialization ends with setting the SPI speed to maximum.

### 5.4 Block (sector) size

The SD card specification includes several different sector sizes, but practically all cards so far seem to use 512 bytes. (Perhaps future bigger SDXC cards will use larger sector sizes?)

For 8-bit microcontrollers with limited RAM, reserving 512 bytes is quite much as it is, which is why this and other libraries impose a support limitation to 512 bytes per sector.

## 5.5 Understanding FAT16 and FAT32

The first sector on the storage medium contains the MBR, Master Boot Record. It holds code for computer booting, the partition table and information about which partition (if any) is the boot partition. For SD cards, there is usually only one partition. Each partition contains a BPB, Boot Parameter Block that stores all the partition-specific data like:

- partition size in sectors
- file system ID
- the number of bytes per sector (only 512 is supported by this library)
- the number of sectors per cluster
- the start, size, and number of FATs (File Allocation Tables)
- the start and size of the FAT16 root directory
- the start of the data area

In some cases when there is only one partition, the above information is stored directly in the MBR.

This library currently only supports one partition, but you can modify it if you need more.

After the SD card initialization, the next step is the file system initialization, in which these and other parameters are read, calculated, and stored in system variables for your application's needs.

## 5.6 Byte in sector, sector in cluster, cluster, and absolute sector

In FAT16 and FAT32, the main addressable chunk of data is called a cluster. It consists of 1 to 64 sectors that each contains 512 data bytes. (Higher sector counts and bigger sectors are covered by Microsoft's FAT specifications, but "number of sectors \* bytes per sector" may not exceed 32k, which gives the above possible values for 512 bytes/sector operation.)

Clusters 0 and 1 are reserved, so the first usable cluster is 2. (Unless it has been damaged, in which case the first usable cluster is higher than 2. This very rare case is not supported by this library.)

If you want to read a particular byte, you need to know the cluster number, the sector number within that cluster, and the byte number within that sector. You must always keep track of these three variables. They are used to calculate the absolute SD card sector number that is used when reading from and writing to the card. When navigating in directories, you also need to know the cluster number in which the current directory starts.

## 5.7 Root directory

The starting point in the file system is the root directory. In FAT16, it is a dedicated sector range (and therefore limited in size (to 512 entries)). It has a volume ID and (unless the partition is empty) a catalog entry for each file and subdirectory. Each entry is 32 bytes long.

The directory entry holds all information about the file's or subdirectory's:

- name
- start cluster
- date, time, and other attributes
- size

FAT32 doesn't have a dedicated root directory. Instead, it's placed in the first usable cluster (typically 2), which means that its size isn't limited like in FAT16.

## 5.8 FAT, File Allocation Table

Every file or subdirectory starts in a unique cluster address in the data area. When the file or subdirectory has filled a cluster, another cluster is used. This forms a chain of clusters and the catalogue of these links is stored in the FAT.

As mentioned above, the file or subdirectory start is pointed out by the directory entry. This cluster is the beginning of the file or subdirectory chain. The cluster number also corresponds to a FAT entry, that either points to the next used cluster or contains an EOC, End Of Chain marker.

In other words: to read a file, you:

1. Must know in which directory cluster chain there is an entry that points to it. This is represented by the (first) cluster number for the particular directory.
2. Compare the name and attributes for each 32-byte directory entry in the first cluster's first sector, then the next sector until you have reached the end of the cluster's last sector.
3. Read the FAT for this cluster in order to find the next cluster.
4. Iterate steps 2 and 3 until you have found the file or reached an EOD (End Of Directory) marker in step 2 or an EOC marker in step 3.
5. If you find the file's directory entry, read the file's start cluster and file size from it.
6. Read from the file, starting at the first cluster's first sector, then the next sector until you have reached the end of the cluster's last sector.
7. Read the FAT for this cluster in order to find the next cluster.
8. Iterate steps 6 and 7 until you have reached the end of the specified file size or (in case of a media or file system error) until you have reached an EOC marker in step 7.

For redundancy reasons, the FAT is usually stored in two separate copies. According to the FAT32 specification, there can be any number from 1 and up. This library supports exactly two FATs (but you can modify this if you like).

## 5.9 Fsinfo

The fsinfo is a FAT32 sector that contains two 32-bit numbers. They hold the number of free clusters and the number of the next not yet used cluster (which is often higher than the first reusable cluster, N.B.). This speeds up certain file system queries (e.g. free space) and if you use this information when writing, you decrease fragmentation which is valuable on hard drives and get a more even media wear.

Earlier Microsoft FAT implementations first try to reuse clusters that previously contained data that is now erased, while later implementations first go to the fsinfo next free cluster.

This library uses the first approach, partly because there is only a very small fragmentation penalty (if any) on SD cards, and partly because FAT16 doesn't keep track of the next unused cluster, which means I only had to write one implementation.

In other words, this library is not using the fsinfo data when preparing for the write/append, but it optionally updates the fsinfo after writing/appending.

However, if you prefer the second approach (using the fsinfo next free cluster value when writing), you can modify the write and append routines.

## 5.10 Long filenames

Originally, DOS used 8.3 filenames; 8 capital name characters and 3 capital extension characters. While this preserves directory space, long filenames are more informative and user-friendly.

In FAT, a long filename can be up to 255 characters with more freedom in terms of valid characters. It is split in 13-character chunks and stored in the directory just before the "real" 8.3 directory entry. This means that a single long filename can take up 21 directory entries, which consumes more card space and also results in much more time-consuming directory searches.

In comparison, this library requires a total of about 598 bytes of RAM for reading, of which 512 is used to store the current absolute sector. Support for long filenames increases the RAM requirement by 256 per filename string. If you want to compare filenames you will need two of these strings.

I came to the conclusion that long filenames are useful in some media implementations like mp3 players, but not required when writing data to the SD card using an AVR microcontroller. However, if you want support for long filename writing it is not very difficult for you to add it.



## 6 Using the library

### 6.1 Configuration

The library consists of two files:

- SPI settings and data declarations. This must be `$INCLUDEd` at the start of your code.
- The actual code. This must be `$INCLUDEd` at the end of your code.

First, make sure that the SPI speed setting (based on your physical clock speed) is in the 100 – 400 kHz range. If necessary, change the pin definitions at the beginning of the declarations. Also, verify the setting to double speed at the end of `Sdinit`.

Next, make the appropriate settings at the top of the declarations file. This is where you control which code is actually going to be used by the compiler. Your main choices are:

- Reading and writing or read only?
- Appending?
- If you are writing, will you buffer the FAT sector for increased large file writing speed?
- Are you using the smaller `FIND` or the scrolling `DIRLIST` routine when searching for files and subdirectories?
- Are you using long filenames when reading? (Requires `DIRLIST`.)
- Are you also updating the `fsinfo` sector when writing? This isn't strictly necessary, so you can trade write speed against file system hygiene.
- Do you need the ability to wipe the card?

### 6.2 Knowing your whereabouts

Regardless of what your application is going to do, you must keep track of:

- the first cluster of the current directory (chain)
- the current cluster, sector number in this cluster, and byte in this sector for:
  - the directory
  - the file you are reading (when writing these variables are used as temp variables)
  - the file you are writing
  - the FAT1 sector number and byte in this sector

Please see the declaration section for a commented listing of these and other variables used in this library.

### 6.3 The root directory

As there is no cluster 0 in FAT16/32, this “dummy” cluster number represents the root directory in this library. For FAT32, you can also specify cluster 2 when addressing the root directory, but for FAT16, there is no other way to address the root.

## 6.4 Creating new files and directories and appending to files

Please note that this library doesn't use the terms "open" and "close". Instead, the gosub names refer to the actual operations they perform.

### 6.4.1 Reading a file

Reading a file has two steps:

- Finding the start cluster for the file (done via `Sdfindentryindirectory` or `Sddirlist`).
- Reading single bytes, typically in a loop (done via `Sdreadbyte`).

As this library doesn't update the directory entry variable for last access date, it doesn't make any changes that need to be written back to the card. For this reason, there is no "close" – just read the last byte you are interested in and then proceed to the next thing you want to do.

### 6.4.2 Creating a new file

To create a new file, you need to call three gosubs in sequence:

1. `Sdcreatefileordir` with parameter `Sdcreatemode = 0`. This creates a directory entry including everything except the file size and sets the whereabouts. If necessary, it also extends the directory cluster chain. `Sdfindentryindirectory` or `Sddirlist` is used to find the available directory entry.
2. `Sdwritebyte` for each byte. It also takes care of data cluster chain extension.
3. `Sdfinalizeafterwriting` to save the last data sector (if necessary), save the EOC marker in the data FAT chain, and store the file size in the directory entry. There is a setting for erasing or not erasing the rest of the sector before writing it to the SD card. (Currently the library doesn't erase the remaining sectors in the current file cluster.)

Please note that if you are updating the `fsinfo` data (`Sdusefsinfo = 1`), you must call `Sdreadfsinfo` (anytime) before calling `Sdcreatefileordir`. (Anytime) afterwards, you must call `Sdwritefsinfo` to save the updated variables back to the SD card.

### 6.4.3 Creating a new subdirectory

To create a new subdirectory, you only need to call `Sdcreatefileordir` with parameter `Sdcreatemode = 1`. This creates a directory entry and if necessary it also extends the directory cluster chain. Finally, it erases the entire new subdirectory cluster and stores the "." and ".." entries at the beginning.

Please note that if you are updating the `fsinfo` data (`Sdusefsinfo = 1`), you must call `Sdreadfsinfo` (anytime) before calling `Sdcreatefileordir`. (Anytime) afterwards, you must call `Sdwritefsinfo` to save the updated variables back to the SD card.

#### 6.4.4 Appending to a pre-existing file

To append to a pre-existing file, you need to call three gosubs in sequence:

1. `Sdpreparetoappend` to locate the directory entry, read file size, go to the last previously written byte, and set the whereabouts. `Sdfindentryindirectory` or `Sddirlist` is used to find the available directory entry.
2. `Sdwritebyte` for each byte. It also takes care of data cluster chain extension.
3. `Sdfinalizeafterwriting` to save the last data sector (if necessary), save the EOC marker in the data FAT chain, and store the file size in the directory entry. There is a setting for erasing or not erasing the rest of the sector before writing it to the SD card. (Currently the library doesn't erase the remaining sectors in the current file cluster.)

#### 6.5 Write safety vs. performance

This library supports 2.5 ways of handling file writing and appending:

- `Sdcreatefileordir/Sdpreparetoappend`, `Sdwritebyte`, and `Sdfinalizeafterwriting` for each byte you are writing to the file. This is the safest way, but it has a performance penalty.
- `Sdcreatefileordir/Sdpreparetoappend`, `Sdwritebyte` until you have written the last byte, after which you `Sdfinalizeafterwriting`. This is the best from a performance perspective, but if you e.g. have a power failure, the data in the last sector gets lost and the file size in the directory becomes wrong, which in many cases appears as a corrupt file. You will be able to salvage the data written to the SD card, at least if you use a tool like WinHex.
- The 3<sup>rd</sup> way is a hybrid solution that is not currently implemented, but it should be fairly easy to accomplish: `Sdcreatefileordir/Sdpreparetoappend`, `Sdwritebyte` and copying `Sdbuffer()` to `Sdyourarray()` just before calling `Sdfinalizeafterwriting`. When you want to append another byte, you copy the file sector back from `Sdyourarray()` to `Sdbuffer()`, then `Sdwritebyte` and `Sdfinalizeafterwriting` again. This is a way to obtain maximum data safety whilst minimizing the performance cost.

#### 6.6 Copying a file or reading and writing concurrently

This is not implemented, but you could modify the library to support it. One approach would be to declare additional "whereabouts" variables, `Sdfile1array()`, `Sdfile2array()` and copy back and forth between these and `Sdbuffer()`.

An alternative approach would be to read 512 bytes at the time from "file 1" into `Sdyourarray()` and then loop through it, appending to "file 2". This would require a second set of many of the "whereabouts" variables.

#### 6.7 Wiping the SD card

By using `Sdwipe`, you can wipe the card. If you set parameter `Sderaseall = 1`, it will write zeros to every sector starting with FAT1 and to the last sector of the partition. If you set this parameter to 0, it will stop after clearing cluster 2. This is not a real formatting as it doesn't generate MBR and BPB, but as long as you haven't written outside of the FAT, directory, or data area, the effect will be the same. Please note that a full wipe takes a long time to perform.

## 6.8 Examples

In the “Main.bas” and “Example\_...” files you can find examples of the most common usage scenarios. Please note that these files also use a Nokia 3310 display library, that I used it when developing and debugging. Unless you will be using this display, you must delete these lines or replace them with whatever your application requires.

### 6.8.1 Initialization sequence

To initialize the SD card and the FAT16/32 filesystem, you need these lines at the beginning of your program:

```
Sdstatus = 0
Gosub Sdinit
If Sdstatus = 0 Then
  ' Initialization successful. Sdcardtype = 1 -> v1.x SDSC, 2 -> v2+ SDSC, 3 -> v2+ SDHC or SDXC
  Gosub Sdinitfs
End If
If Sdstatus = 0 Then
  ' Now the card is ready for file system operations.
  ' Place your code here.
Endif
```

### 6.8.2 Code size and RAM usage for some of the examples

Functionality	Pgm	RAM	\$hwstack
Read and write block (no file system)	2204	532	4
Find and read FAT16/32	5678	588	~10
Dirlist to locate certain file and then read FAT16/32 8.3	6856	643	~10
Dirlist to locate certain file and then read FAT16/32 LFN	7774	1189	~10
Dirlist to scroll FWD+BWD and print entry names FAT16/32 8.3	6888	643	~10
Dirlist to scroll FWD and print entry names FAT16/32 LFN	7576	933	~10
Writing with find, read, write FAT16/32 with fsinfo, unbuffered FAT sector	11786	651	~10
Writing with find, read, write FAT16/32 without fsinfo, unbuffered FAT sector	11374	639	~10
Writing with find, read, write FAT16/32 with fsinfo, buffered FAT sector	12646	1163	~10
Writing with find, read, write FAT16/32 without fsinfo, buffered FAT sector	12204	1151	~10
Writing with dirlist, read, write FAT16/32 with fsinfo, unbuffered FAT sector	12734	699	~10
Writing with dirlist, read, write FAT16/32 without fsinfo, unbuffered FAT sector	12330	687	~10
Writing with dirlist, read, write FAT16/32 with fsinfo, buffered FAT sector	13584	1211	~10
Writing with dirlist, read, write FAT16/32 without fsinfo, buffered FAT sector	13150	1199	~10
Appending with dirlist, read, write FAT16/32 with fsinfo, buffered FAT sector	14294	1211	~10

## 6.9 Sdstatus

Sdstatus is the main status variable. In most cases, 0 means that the operation went well (but not always). Certain gosubs require that you first make sure that Sdstatus = 0.

## **7 General information about BASCOM AVR's use of RAM**

At the end of this application note, I would like to include information about BASCOM AVR's use of RAM. This is because most other SD card libraries seem to use function calls, while my library is using gosub routines in order to reduce auxiliary RAM usage in the software stack and frame.

Most of the below is copied from a [www.mcselec.com](http://www.mcselec.com) user forum message by DToolan in 2006 and the rest of it is revised by Mark Alberts. If you want to go even deeper, please also read AN #183 - "Reveal the secret of Stack" BASCOM-AVR - Part 1.

### **7.1 Hardware Stack**

Stores the present program counter (return address within a program) for when a GOSUB or CALL routine is executed. This is so it knows where to go back to after it has finished each call.

Each GOSUB or CALL routine uses 2 bytes to store the present program counter.

Nested GOSUB / CALL routines (calling one subroutine which calls another) will use 2 additional bytes for each CALL / GOSUB.

A recursive function/procedure will use an unknown number of hardware stack data.

An interrupt service routine can use up to 32 bytes of the hardware stack, but this is a worst case scenario. Built-in commands that use interrupts, use fewer registers.

In the worst case scenario, it will store the present values of most CPU registers (R0 ~ R31) and SREG (the CPU status register).

Processors that have a RAMPZ register, will save the state of the RAMPZ register as well.

### **7.2 Software Stack**

The software stack stores the addresses of variables that are passed to a subroutine and any LOCAL variables within that routine.

Each passed variable and local variable uses 2 bytes each for their respective addresses.

If you have used 10 locals in a SUB and there are 3 parameters passed to it, you need  $13 * 2 = 26$  bytes.

Some statements/functions with more than 2 parameters will also use 1 or 2 bytes. In most cases, values will be passed in registers for internal function/statements.

### 7.3 Frame

This is temporary storage space; a scratch pad for functions or commands to doodle on.

Local variables are stored there and if you pass variables BYVAL, you will also use the frame space.

When you have 2 local integers and a string with a length of 10, you need a frame size of  $(2*2) + 11 = 15$  bytes.

Please notice that strings (always) take up one more byte than the declared length (for storing the end of string marker).

When your string was dimmed with a length of 20, it takes up 21 bytes of the frame space.

As of BASCOM AVR 1.11.8.2, the actual frame size is 24 bytes less than the declared value.

These 24 bytes are partitioned aside for use by Bascom string handling routines, conversion routines and other functions.

The frame space requirement depends on which conversion routines are actually used by the developer.

For example, INPUT num, STR(), VAL(), FORMAT(), etc need a maximum of 16 bytes, while floating point conversion requires more than 16 bytes.

For this reason, the recommended default \$FRAMESIZE = 16 actual + 24 sectioned away by Bascom (reserved) = 40 (recommended minimum bytes declared)

In the example above, you need a  $15 + 24 = 39$  frame size, so you might just as well stick to the default value of 40.

When adding strings you could also need frame space. See this example:

```
Dim s as string * 20
```

```
S="abcdefgh"
```

```
S=str(var) + "text" + s
```

In this case the old value of S is used so it needs to be saved in the frame so it can be added.

In the above example you need 21 bytes for local variable S plus space for its temporary copy.

## 7.4 RAM organization

The hardware stack, software stack and frame are always located at the top of memory and grow downward (when you expand their sizes). As a user you cannot control the direction.

The hardware stack and software stack are then utilized from their highest memory locations downward while the frame is used from its lowest memory location upward.

BASCOM will let you know if you have allocated so much space to your hardware stack, software stack and frame that they have encroached upon your dimmed variable space.

This equates to "out of ram".

M162 example:

```
'+-----+ Top of sram 0x04FF
'|
'| Hardware  |
'|   stack   |
'|+-----+
'| Software  |
'|   stack   |
'|+-----+
'|   Frame   |
'|+-----+
'|  24 Byte   | <- subtracted from declared $FRAMESIZE
'|Frame Reserved|
'|-----|
'| Unused SRAM |
'|           |
'|+-----+
'| User Dimmed |
'| Variables   |
'|           |
'|+-----+ Bottom of sram 0x0100
```

However, BASCOM is considering another way to organize the RAM in the future:

"If the frame is located after the dimmed variables, the SW stack and frame can walk into each other. \$swstack could be omitted in that case. It needs a change in the assembler/compiler. For many users it will be more convenient."