CODE TIME TECHNOLOGIES

Abassi RTOS

SD/MMC Support

Disclaimer

Code Time Technologies Inc. provides this document “AS IS” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the SD/MMC driver used by Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). A standalone version of the SD/MMC driver is also described in here.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`???_sdmmc.h`	Include file for the SD/MMC driver (??? is target dependent)
`???_sdmmc.c`	"C" file for the Abassi SD/MMC driver (??? is target dependent)
`???_sdmmc_CMSIS.c`	"C" file for the CMSIS RTOS API SD/MMC driver (??? is target dependent)
`SAL.h`	Include file for the standalone abstraction layer (supplied with standalone package only)
`SAL.c`	"C" file for the standalone abstraction layer (supplied with standalone package only)
`ISRhandler_???.s`	"ASM" add-on file for the standalone version only. It contains support for both the driver and the demo application.
`Demo_9_<PROC>_<TOOL>.c`	SD/MMC small shell demo based on FatFS. *<TOOL>* is the name of the build environment tool-set, *<PROC>* is the processor/target.
`Demo_2`***n***`_<PROC>_<TOOL>.c`	SD/MMC small shell demo based using the system call layer with different files system stacks. *<TOOL>* is the name of the build environment tool-set, *<PROC>* is the processor/target. ***n*** is 0 to 9.

1.2 Limitations

The standalone version is the same as the RTOS version except it uses an adaptation layer to emulate the RTOS functionality. The standalone version of the driver does not use or support interrupts. All wait operations use polling. Please refer to the specific driver code for a description / list of these limitations; this is described near the top of the files `??_sdmmc.h` and `??_sdmmc.c`.

1.3 Features

The SD/MMC driver API is kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself and embedded comments.

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and µAbassi.

2 Target Set-up

All there is to do to configure and enable the use of the SD/MMC driver in an application based on Abassi is to include the following file in the build:

- `???_sdmmc.c` (For Abassi & standalone)
- `???_sdmmc_CMSIS.c` (For CMSIS RTOS API)
- `SAL.c` (For Standalone)
- `ISRhandler_???.s` (For Standalone)

and set-up the include search directory order making sure the file `???_sdmmc.h` is found (and `SAL.h` for the standalone)

The SD/MMC driver may or may not, depending on the target platform, be independent from other include files.

2.1 Build Options

There are a few build options that allow the SD/MMC driver to be configured for the needs of the target application. The following table lists all of them:

Table 2-1 Build Options

File Name	Default	Description
`OS_PLATFORM`	`Platform specific`	Number indicating the target platform. Refer to `???_sdmmc.h` to see the list of supported platforms and the default one.
`SDMMC_CLK`	`Target dependent`	Clock frequency of the SDMMC cotroller.

`SDMMC_BUFFER_TYPE`	`SDMMC_BUFFER_UNCACHED`	Type of buffering:
		`SDMMC_BUFFER_UNCACHED`
		`SDMMC_BUFFER_CACHED`

`SDMMC_NUM_DMA_DESC`	`Platform specific`	Number of DMA buffers internally used by the SD/MMC driver
`SDMMC_REMAP_LOG_ADDR`	`1`	Boolean to enable/disable the conversion from logical to physical address
`SDMMC_DEBUG`	`0`	Boolean controlling the sending of progress / debug messages to `stdout`.
`SDMMC_ARG_CHECK`	`1`	Boolean to enable/disable the check on the validity of the API function arguments

Table 2-2 Build Options (RTOS only)

File Name	Default	Description
`SDMMC_USE_MUTEX`	`0`	Boolean controlling if a dedicated mutex is used or not for exclusive access to the driver.

2.1.1 OS_PLATFORM

The build option `OS_PLATFORM` informs the SD/MMC driver of which platform it is operating on. It is used to know the base address of the SD/MMC peripheral modules and type of modules.

2.1.2 SDMMC_CLK

The build option `SDMMC_CLK` defines the clock frequency the SDMMC controller operates with. A default value is set according to the target platform specified by `OS_PLATFORM`. If the module clock frequency is different from the default value, all there is to do is defined the build option `SDMMC_CLK` and set it to the clock frequency in Hz.

2.1.3 SDMMC_BUFFER_TYPE

The build option `SDMMC_BUFFER_TYPE` sets the SD/MMC driver to either use cached buffers or non-cached buffers. When set to `SDMMC_BUFFER_UNCACHED`, the internal SD/MMC DMA buffers are located in the "`.uncached`" linker section and this data section must absolutely not be cached. When the build option is set to `SDMMC_BUFFER_CACHED`, then the internal SD/MMC DMA buffers are located in the default data memory, which would be typically cached but can also be uncached.

2.1.4 SDMMC_NUM_DMA_DESC

The SD/MMC deriver relies on internal buffers used by the SD/MMC peripheral modules. This build option should never be modified but is necessary to be exported as it is used to inform the application about the maximum buffer that can be transferred by the SD/MMC driver. The maximum buffer size, read or written, is `512 * SDMMC_NUM_DMA_DESC` bytes. If more than this size is requested to be read/written, the request will be aborted with the report of an error.

2.1.5 SDMMC_REMAP_LOG_ADDR

When the MMU is set-up to remap memory areas at different addresses from the physical address, it is necessary to convert the logical address to their physical equivalents because DMA transfers are use to transfer the data from/to SDMMC and Memory. The build option `SDMMC_REMAP_LOG_ADDR` is a Boolean that selects if the addresses used by the DMA are converted from logical to physical. By default it is set to a non-zero value (enable). Although the remapping function is a low instruction count, one may want to not perform a redundant remapping when the logical addresses are the same as the physical. The remapping can be turned off setting the build option `SDMMC_REMAP_LOG_ADDR` to zero.

2.1.6 SDMMC_DEBUG

The build option `SDMMC_DEBUG` controls the printout of progress and error messages to `stdout`. This build option is a Boolean; when set to a non-zero value, messages will be sent to `stdout`, and when set to a value of zero, messages will not be sent to `stdout`.

2.1.7 SDMMC_ARG_CHECK

The build options `SDMMC_ARG_CHECK` controls if the driver checks the validity of the API function arguments or not. This build option is a Boolean; when set to a non-zero value, the driver checks the validity of the arguments and returns an error code when the arguments are invalid. When set to a zero value, it does not check the validity of the arguments.

2.1.8 SDMMC_USE_MUTEX

In an RTOS environment (either Abassi or the CMSIS RTOS API), the driver can internally protect access to the SD/MMC peripheral module through a mutex, using one mutex per module. Setting the build option `SDMMC_USE_MUTEX` to a non-zero value will make the driver used a dedicated internal mutex. Setting the build option `SDMMC_USE_MUTEX` to a zero value will make the driver not use a dedicated internal mutex. When set to zero, the application or file system stack must handle the exclusive access to the driver if more than one task can access it.

3 API

In this section, the API of all common SD/MMC driver functions is provided. The next section gives examples on how to use the SD/MMC.

3.1.1 mmc_init

Synopsis

```
#include "???_sdmmc.h"

int mmc_init(int Dev);
```

Description

`MMCinit()` is the component used to initialize the SD/MMC module driver. The device's controller number to initialize is indicated by the argument `Dev`.

Arguments

`Dev`	Device's controller number (Number starting at 0)

Returns

`int`	`== 0` : success
	`!= 0` : error

There are many scenarios that could make the SD/MMC initialization fail (return value != 0). The return value does not indicate the cause of the failure, but turning on the debug will provide the root cause of the failure. The possible causes of failure are:

- The reset of the peripheral module failed
- The SD/MMC module is not responding
- The SD/MMC cannot reach the Idle state
- A command prefix sent triggered an error
- The SD/MMC did not respond to the OCR request
- The SD/MMC card cannot be used

Component type

Function

Options

Notes

See Also

3.1.2 mmc_sendcmd

Synopsis

```
#include "???_sdmmc.h"

int mmc_sendcmd(int Dev, unsigned int Cmd, uint32_t Arg,
                uint32_t Exp, uint32_t *Resp, MMCdata_t *Data);
```

Description

`mmc_sendcmd()` is the component used to perform all operational requests with the SD/MMC modules. The device's controller number is indicated by the argument `Dev`. The argument `Cmd` is one of the standard MMC commands. Depending on the command, an argument is most likely required. When that is the case, then the command argument is supplied through the argument `Arg`. Each command has a specific type of response when successful, and this is provided by the argument `Exp`. When the value of the response to the command is needed by the application, it can be extracted from the argument `Resp`. Finally, the argument `Data` is the buffer to hold/supply data to be exchanged with the driver or SD/MMC card.

Arguments

`Dev`	Device's controller number (Number starting at 0)
`Cmd`	Command to send to the SD/MMC card (See section 5.1)
`Arg`	Commend specific argument
`Exp`	Expected response (See section 5.2)
`Resp`	Response from the card. Can use `NULL` if the value if not required

`Data`	When data is exchanged with the card, pointer to the exchange buffer.
	This is the data structure of type `MMCdata_t` with the three following fields:
	`char *Buffer` : data to exchange
	`unsigned int Flags` : set to `MMC_DATA_READ` or to `MMC_DATA_WRITE`
	`unsigned int Nbytes` : number of bytes in `Buffer`

Returns

`int`	`== 0 :` the command was successful `!= 0 :` the command failed

Component type

Function

Options

Notes

An understanding of the SD/MMC protocol is needed to properly use the `MMCsendCmd()` component. No further details are provided as this is outside the scope of this document, but the reader can refer to section 4 for some basic usage examples. Also, the standards are available from [R4] and [R5].
The SD/MMC drivers are always supplied with the code interfacing the FatFS file system stack [R6] with the SD/MMC driver. This code may be used as a springboard to interface with other file systems stacks.

See Also

3.1.3 mmc_sendstatus

Synopsis

```
#include "???_sdmmc.h"

int mmc_sendstatus(int Dev);
```

Description

`mmc_sendstatus()` is the component used to send a status request to the card. The device's controller number is indicated by the argument `Dev`.

Arguments

`Dev`	Device's controller number (Number starting at 0)

Returns

`int`	`== 0 :` success, the card is ready to transfer new data
	`!= 0 :` error, no reply from the card or it is not ready

Component type

Function

Options

Notes

See Also

3.1.4 mmc_present

Synopsis

```
#include "???_sdmmc.h"

int mmc_present(int Dev);
```

Description

`mmc_present()` is the component that reports if a card is present in the socket. The device's controller number is indicated by the argument `Dev`.

Arguments

`Dev`	Device's controller number (Number starting at 0)

Returns

`int`	`== 0 :` no card is inserted
	`!= 0 :` a card is inserted

Component type

Function

Options

Notes

The return value is always non-zero (card present) for mini- and µSD cards as there is no pin in the mini- and µSD card electrical interface for this information. Therefore for mini- and µSD cards, it will report a card is present even if there is no card inserted.

See Also

3.1.5 mmc_nowrt

Synopsis

```
#include "???_sdmmc.h"

int mmc_nowrt(int Dev);
```

Description

`Mmc_nowrt()` is the component that reports if the card inserted in the socket is write protected. The device's controller number is indicated by the argument `Dev`.

Arguments

`Dev`	Device's controller number (Number starting at 0)

Returns

`int`	`== 0 :` the card is write protected `!= 0 :` the card is write enabled

Component type

Function

Options

Notes

The return value is always zero (card is write enabled) for mini- and µSD cards as there is no write enable/disable switch on these cards.

See Also

3.1.6 MMCintHndl_*n*

Synopsis

```
#include "???_sdmmc.h"

void MMCintHndl_n (void);
```

Description

`MMCintHndl_n()` is the interrupt handler for the SD/MMC driver for the device controller #***n*** (not used by the standalone version).

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

If the interrupt handler is not attached and the related interrupt enabled, the SD/MMC driver will not operate at all (not applicable for the standalone version).

See Also

3.1.7 Exported Variables

Some variables are exported by the SD/MMC driver in order to provide more information about the inserted card. All exported variable are arrays and the array indexing is the controller device number, therefore for device N, the associated value is located in the array at index N. The following table lists and describes them

Table 3-1 Build Options

Variable	Data Type	Description
`G_MMCblkLen[]`	`unsigned int`	Block length (in bytes) used by the inserted card
`G_MMCcapacity[]`	`uint64_t`	Capacity (in bytes) of the inserted card
`G_MMCcardOCR[]`	`uint32_t`	Operating Condition Register (OCR) of the card. OCR register bit position #0 of the OCR is the LSBit in `G_MMCcardOCR[]`.
`G_MMCscr[][8]`	`char`	8 bytes holding the SD Configuration Register (SCR) of the card. The contents of `G_MMCscr[0]` is SCR slice 63:56 and `G_MMCscr[7]` is SCR slice 7:0.
`G_MMCrca[]`	`uint32_t`	Relative Card Address (RCA) used by the card

Although these exported variables are read-write, their contents should never be modified as it could lead in the misbehavior of the SD/MMC driver. In other words, these should be used as read-only variables.

4 Examples

4.1 Initialization

The first step required when using the SD/MMC driver is to reset and do the basic initialization of the SD/MMC device controller; here we consider `Drv1` (device #1) as the SD/MMC controller number. This needs to be done only once and it must be performed before using any SD/MMC driver components. The interrupt handler can only be used with Abassi, and with Abassi it must be used. With the standalone version, do NOT attach/use the interrupt handler or enable the related interrupt.

Table 4-1 SD/MMC initialization

```
…

if (0 != mmc_init(Drv1)) {
    printf("MMC init error\n");
}

OSisrInstall(SDMMC_INT, MMCintHndl_1);/* Install SD/MMC driver interrupt handler   */
GICenable(SDMMC_INT, 128, 0);         /* This is for Drv1 which is device #1       */

…
```

4.2 SD/MMC write

The following example shows how to write a buffer `Buff`, holding a number `Nsect` of 512 bytes sectors and starting the writing at sector number `SectNum`. This example considers the buffer to be located in cached memory, therefore with the build option `SDMMC_BUFFER_TYPE` set to `SDMMC_BUFFER_CACHED`:

Table 4-2 SD/MMC write

```
int MMCwrite(int Drv, const char *Buff, int Nsect, int SectNum)
{
int          ii;                        /* General purpose                          */
MMCdata_t    MMCdata;                   /* MMC DMA descriptor                       */
const char  *PtrC;                      /* Pointer in the buffer to read            */
uint32_t     SectNow;                   /* Current sector to read                   */
int          Size;                      /* Number of sectors to read                */

   MMCdata.Flags   = MMC_DATA_WRITE;     /* Always writing to the SD/MMC             */
   PtrC            = Buff;               /* Start at the beginning of the input buffer*/
   while (Nsect > 0) {
       Size = Nsect;                     /* Sectors & SDMMC blocks size are 512 bytes */
       if (Size > (SDMMC_NUM_DMA_DESC)) {/* The maximum transfer size also limited by */
           Size = (SDMMC_NUM_DMA_DESC);  /* the sd/mmc driver internal number of DMA  */
       }                                 /* descriptors                              */
       MMCdata.Nbytes = Size<<9;
       MMCdata.Buffer = (void *)PtrC;    /* Select the destination buffer            */

       Nsect    -= Size;                 /* This less number of sectors to write     */
       SectNow   = SectNum;              /* Current sector to write                  */
       SectNum  += Size;                 /* Next sector to write                     */

       PtrC += Size << 9;                /* Adjust the pointer for the next time     */

       if ((G_MMCcardOCR[Drv] & OCR_HCS) != OCR_HCS) {
           SectNow <<= 9;
       }
                                         /* Pre-erase would speed things up          */
       ii = mmc_sendcmd(Drv, (Size > 1) ? MMC_CMD_WRITE_MULTIPLE_BLOCK
                                        : MMC_CMD_WRITE_SINGLE_BLOCK,
                        SectNow, MMC_RSP_R1, NULL, &MMCdata);
       if (ii != 0) {
           PUTS("Failed to write data.");
           return(ERROR);
       }

       if (Size > 1) {                   /* Stop transmission when multi-block       */
           ii = mmc_sendcmd(Drv, MMC_CMD_STOP_TRANSMISSION, 0, MMC_RSP_R1b, NULL, NULL);
           if (ii != 0) {
               PUTS("Failed to stop transmission.");
               return(ERROR);
           }
       }

       ii = mmc_sendstatus(Drv);         /* Waiting for the ready status             */

       if (ii != 0) {                    /* Needed because although the data has     */
           PUTS("Could not get Status.");/* been accepted by the SD/MMC, it takes a  */
           return(ERROR);                /* while for the Programming to complete    */
       }
   }

   return(OK);
}
```

4.3 SD/MMC read

The following example shows how to read into buffer `Buff`, a number `Nsect` of 512 bytes sectors and starting the reading at sector number `SectNum`. This example considers the buffer to be located in cached memory, therefore with the build option `SDMMC_BUFFER_TYPE` set to `SDMMC_BUFFER_CACHED`:

Table 4-3 ISD/MMC read

```
int MMCread(int Drv, char *Buff, int Nsect, int SectNum)
{
int       ii;                           /* General purpose                          */
MMCdata_t MMCdata;                      /* MMC DMA descriptor                       */
char     *PtrC;                         /* Pointer in the buffer to fill            */
uint32_t  SectNow;                      /* Current sector to read                   */
int       Size;                         /* Number of sectors to read                */


   MMCdata.Flags   = MMC_DATA_READ;     /* Always reading from the SD/MMC           */
   PtrC            = buff;              /* Start at beginning of the input buffer*/
   while (Nsect > 0) {
       Size = Nsect;                    /* Sectors and SDMMC block size are 512 bytes*/
       if (Size > (SDMMC_NUM_DMA_DESC)) {/* The maximum transfer size is limited by   */
           Size = (SDMMC_NUM_DMA_DESC);  /* the sd/mmc driver internal number of DMA  */
       }                                /* descriptors                              */
       MMCdata.Nbytes = Size<<9;
       MMCdata.Buffer = PtrC;           /* Select the destination buffer            */

       Nsect    -= Size;                /* This less number of sectors to read      */
       SectNow  = SectNum;              /* Current sector to read                   */
       SectNum  += Size;                /* Next sector to read                      */

       if ((G_MMCcardOCR[Drv] & OCR_HCS) != OCR_HCS) {
           SectNow <<= 9;
       }

       ii = mmc_sendcmd(Drv, (Size > 1) ? MMC_CMD_READ_MULTIPLE_BLOCK
                                        : MMC_CMD_READ_SINGLE_BLOCK,
                        SectNow, MMC_RSP_R1, NULL, &MMCdata);
       if (ii != 0) {
           PUTS("Failed to read data.");
           return(ERROR);
       }

       PtrC += Size << 9;               /* Adjust the pointer for the next time     */

       if (Size > 1) {                  /* Stop transmission when multi-block       */
           ii = mmc_sendcmd(Drv, MMC_CMD_STOP_TRANSMISSION, 0, MMC_RSP_R1b, NULL, NULL);
           if (ii != 0) {
               PUTS("Failed to stop transmission.");
               return(ERROR);
           }
       }
   }

   return(OK);
}
```

5 Appendices

5.1 Commands

The following table identifies the commands to send to the SD/MMC card. If a non-supported command must be used, then using the numerical value of the command will most likely work perfectly.

Table 5-1 SD/MMC commands & responses

Token	Value	Resp	Description
MMC_CMD_GO_IDLE_STATE	0	None	Resets the card to Idle state
MMC_CMD_SEND_OP_COND	1		Initiate initialization process
MMC_CMD_ALL_SEND_CID	2	R2	Ask the card to report the CID numbers
MMC_CMD_SET_RELATIVE_ADDR	3	R6	Ask the card to report the relative address (RCA)
MMC_CMD_SET_DSR	4	None	Program the DSR of the card
MMC_CMD_SWITCH	6		Change SD/MMC card (not used)
MMC_CMD_SELECT_CARD	7	R1b	Toggle the card between stand-by and transfer modes
MMC_CMD_SEND_EXT_CSD	8	R7	Send to the card the external interface conditions
MMC_CMD_SEND_CSD	9	R2	Send the card card-specific data (CSD)
MMC_CMD_SEND_CID	10	R2	Request the card to send its card ID on the CMD line
MMC_CMD_STOP_TRANSMISSION	12	R1b	Stop the card from transmitting
MMC_CMD_SEND_STATUS	13	R1	Request the card to report its status
MMC_CMD_SET_BLOCKLEN	16	R1	Set the block length (in bytes) for all further transfers
MMC_CMD_READ_SINGLE_BLOCK	17	R1	Read a single block
MMC_CMD_READ_MULTIPLE_BLOCK	18	R1	Read multiple blocks
MMC_CMD_WRITE_SINGLE_BLOCK	24	R1	Write a single block
MMC_CMD_WRITE_MULTIPLE_BLOCK	25	R1	Write multiple blocks
MMC_CMD_ERASE_GROUP_START	35	R1	Set the address of the first block to erase
MMC_CMD_ERASE_GROUP_END	36	R1	Set the address of the last block to erase
MMC_CMD_ERASE	38	R1b	Perform the erase operation
MMC_CMD_APP_CMD	55	R1	Inform the card the next command is an app-specific command and not a standard one
MMC_CMD_SPI_READ_OCR	58	R3	Request the card to report the OCR
MMC_CMD_SPI_CRC_ON_OFF	59	R1	Control if CRC is checked or not

5.2 Expected Responses

The following table identifies the expected responses:

Table 5-2 SD/MMC expected response types

Token	Response
MMC_RSP_NONE	None
MMC_RSP_R1	R1
MMC_RSP_R1b	R1b
MMC_RSP_R2	R2
MMC_RSP_R3	R3
MMC_RSP_R4	R4
MMC_RSP_R5	R5
MMC_RSP_R6	R6
MMC_RSP_R7	R7
MMC_RSP_PRESENT	Card is present
MMC_RSP_136	136 bit response
MMC_RSP_CRC	Expect valid CRC
MMC_RSP_BUSY	Card may send busy
MMC_RSP_OPCODE	Response contains op-code

6 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com
[R4] JEDEC, documentation available at http://www.jedec.org
[R5] SD Association, documentation available at http://www.sdcard.org
[R6] FatFS open-source FAT32 stack, available at http://www.elm-chan.org