Getting started with the SPC58EHx/SPC58NHx multimedia card host controller

Introduction

This application note describes the 3MCR (Multi-Card Reader) host interface specific to the SPC58EHx/SPC58NHx microcontroller and explains how to use the module to transfer data from/to SD, MMC and eMMC memory cards in different configurations.

The SPC58EHx/SPC58NHx automotive microcontroller embeds the 3MCR controller that is an advanced host controller compliant with the following specifications:

- MMC specification version 4.51
- SD host controller standard specification version 3.00
- SDIO card specification version 3.00
- SD memory card specification version 3.01
- SD memory card security specification version 1.01

1 3MMC host controller

The 3MCR controller supports several features and provides different programming approaches, for instance the IO method and the DMA data transfer method. The host processor transfers data using the Buffer Data Port register when the IO method is used.

The application will focus on the DMA usage, because the DMA allows the 3MCR to read or write memory without any intervention from the CPU.

The host controller supports two different DMA modes: SDMA and ADMA2 as detailed in the next chapters.

The 3MCR controller handles the eMMC protocol at transmission level, packing data, adding cyclic redundancy check (CRC), start/end bit, and checking for transaction format correctness.

The eMMC card clock frequency is 50 MHz and both the single data rate (SDR) and the dual data rate (DDR) modes are supported. This guarantees the best performance during the transfer of data to the external device.

Concerning the SD/SDIO Card interface, the host clock rate can range between 0 and 50 MHz. Data is transferred in 1-bit and 4-bit SD modes, and the SPI mode is supported, too.

The host controller supports both default-speed and high-speed card. For high-speed cards the controller should clock out the data at the rising edge of the clock, instead for full-speed cards it should clock out the data at the falling edge.

1.1 Register map

The following table reports the address space where the controller registers are mapped on this microcontroller series.

The register space includes the registers used for programming the host controller and an extra subset of registers used to configure the controller it in this platform, for example to enable or disable the feedback clock or the ADMA2 mode. Some extra debug registers are available as described in the reference manual (see Section Appendix C Reference documents).

Refer to the APIs described in Section Appendix A Host controller to access to the space.

Table 1. **Controller register map**

Address range	Registers	Description
0x91000000 0x910000FF	eMMC register map	Host controller registers
0x91000200 0x91000233	eMMC wrapper registers	Wrapper registers to enable internal feature: e.g. feedback clock, ADMA2 and getting extra debug status

1.2 Operating voltage

The signal voltage supported is 3.3 V. To operate at this voltage, the following setting must be made in the digital interface of the power management controller regarding the voltage selection of the IO register:

```
PMCDIG.VSIO.B.VSIO_EMMC = 0;
```

1.3 Bus speed modes

The following table shows the bus speed mode supported for SD and MMC devices.

Table 2. **Bus speed mode supported**

Bus speed mode	Data lines	Frequency	Signal voltage	Bus maximum performance
Secure Digital Card (SD Specifications Part 1 Physical Layer Simplified Specification Version 3.01)				
Default-speed (DS)	4	25 MHz	3.3 V	12.5 MB/sec
High-speed (HS)	4	50 MHz	3.3 V	25MB/sec
Multi Media Card (JESD84-B45)				
Backwards Compatibility with legacy MMC card	1 - 8	0-25 MHz	3.3 V	25 MB/sec
High-speed SDR	4 - 8	0 - 50 MHz	3.3 V	50 MB/se
High-speed DDR	4 - 8	0 - 50 MHz	3.3 V	100 MB/sec

For SD cards, neither SDR nor DDR modes are supported because they need 1.8 V signaling.

1.4 Reset

To reset the peripheral the eMMC_RST must be set in the Reset Generation Module (MC_RGM) register: RGM_PRST2. The 3MMC as peripheral is not actually listed in the MC_ME.PSx registers, because it is considered always active by design.

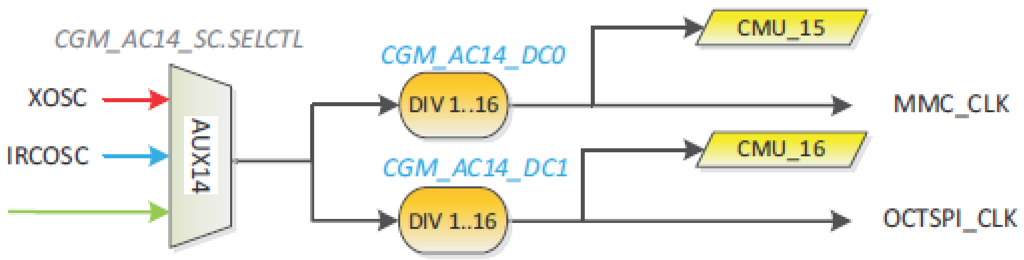
As reported in the device's reference manual (see Section Appendix C Reference documents), some specific parameters for this controller can be tuned in the Platform Configuration Module 3 register (PCM3). The default value of this register guarantees the correct usage of the controller embedded in this architecture resolving unaligned bus transactions and byte swap management. The user should never change the default values of this register.

The controller also provides its own software reset register to perform a software reset of all the data or command lines, or of all the controller excluding the card detection circuit.

1.5 Clocking

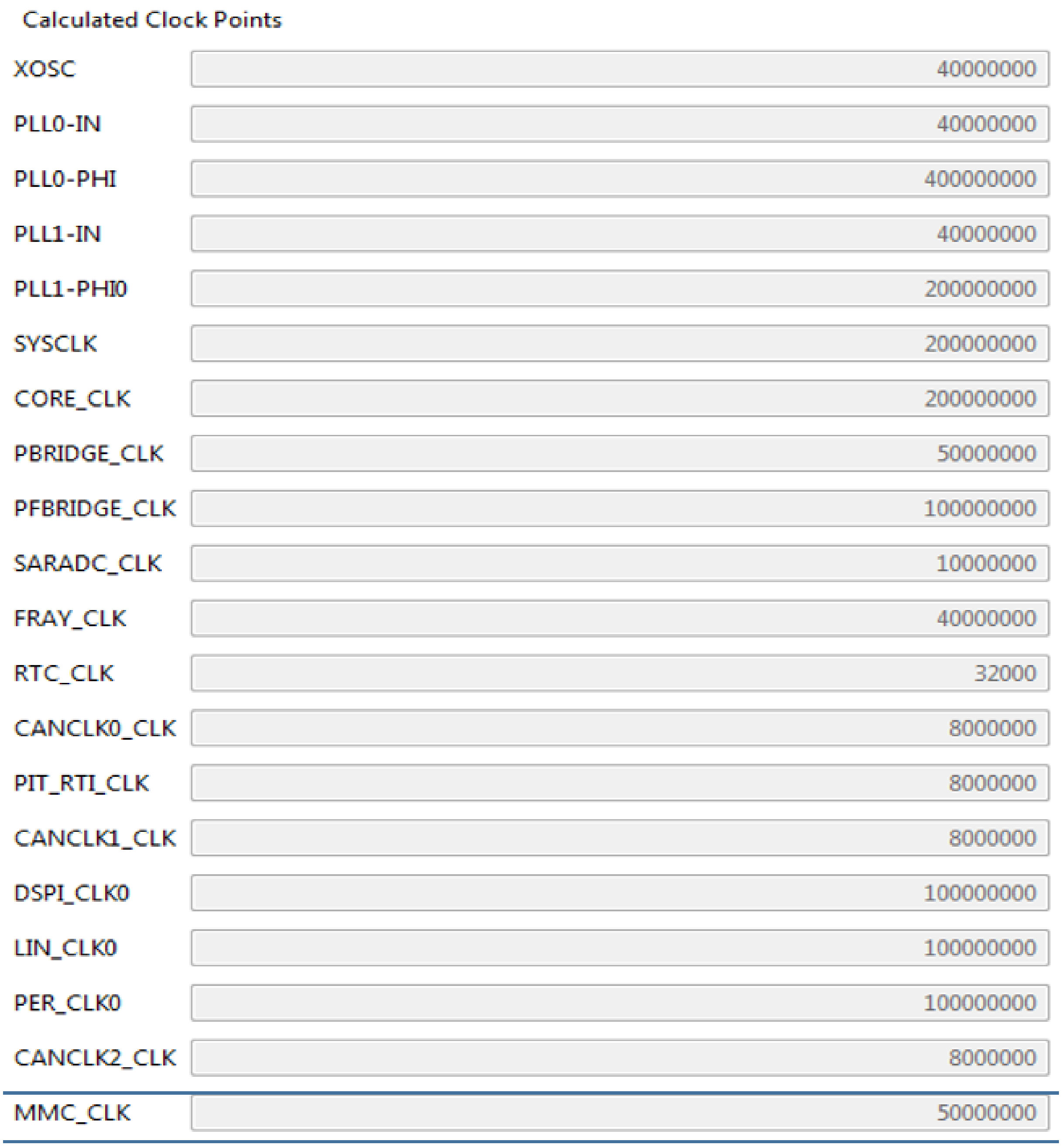
The MMC_CLK is the main clock used for the eMMC interface side of the host controller. This is managed by CGM_AC14_DC0 and CGM_AC14_SC.

Figure 1. **MMC_CLK routing**



The MMC_CLK can be configured using the graphic interface of the SPC5Studio tool, as shown below:

Figure 2. **MMC_CLK view in SPC5Studio**



Calculated Clock Points

XOSC	40000000
PLL0-IN	40000000
PLL0-PHI	400000000
PLL1-IN	40000000
PLL1-PHI0	200000000
SYSCLK	200000000
CORE_CLK	200000000
PBRIDGE_CLK	50000000
PFBRIDGE_CLK	100000000
SARADC_CLK	10000000
FRAY_CLK	40000000
RTC_CLK	32000
CANCLK0_CLK	8000000
PIT_RTI_CLK	8000000
CANCLK1_CLK	8000000
DSPI_CLK0	100000000
LIN_CLK0	100000000
PER_CLK0	100000000
CANCLK2_CLK	8000000
MMC_CLK	50000000

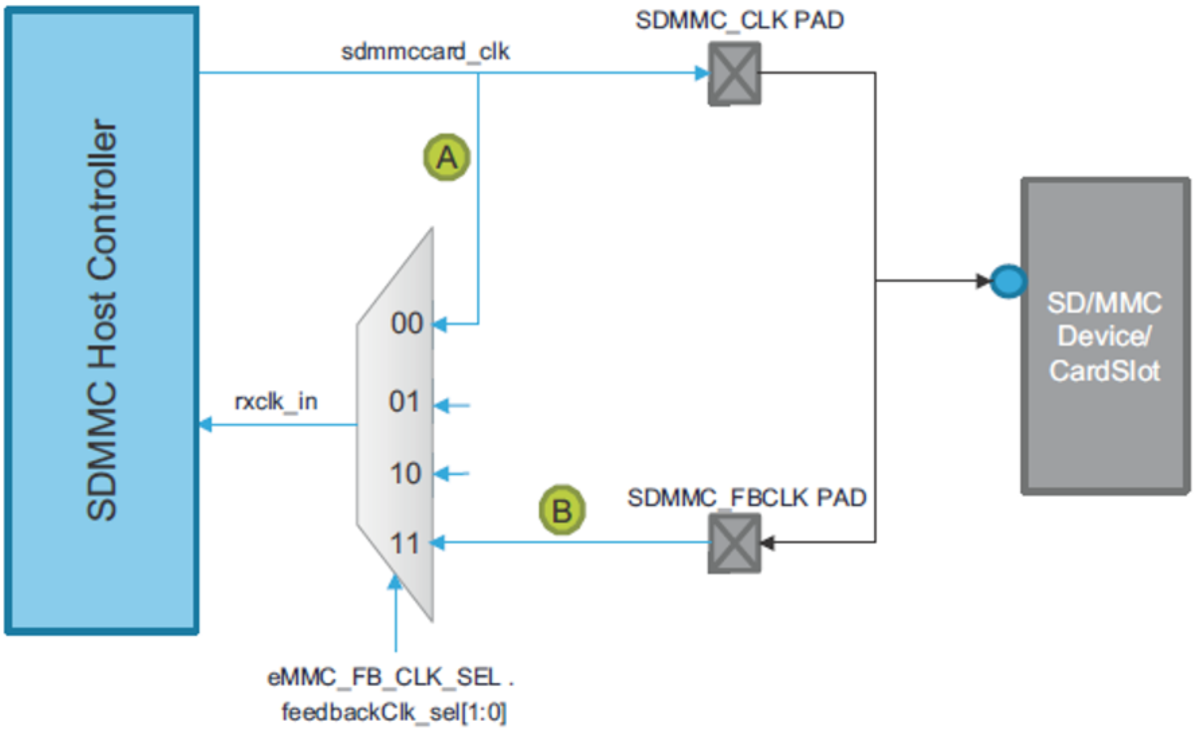
This MMC_CLK clock input is used to generate sdmmcctrl_clk and sdmmccard_clk clocks.

The sdmmcctrl_clk is used by most of the control logic: commands, transmit control and interface side of the block buffer. The sdmmcctrl_clk range is from 400 KHz to 50 MHz.

The sdmmccard_clk is available only when the clock enable is set by the driver and it is supplied to the card.

A feedback clock is also available and is looped to the rxclk_in either from the sdmmccard_clk or from the clock PAD of external memory device. The feedback clock is mandatory to switch on DDR mode.

Figure 3. Feedback clock schema



The following code is used to enable the feedback clock in the FB_CLK_SEL (Feedback clock selection register).

```
Status = sdmmc_ReadReg32(host->Config.BaseAddress, ST_EMMC_FB_CLK_SEL);
Status |= 0x3;
sdmmc_WriteReg32(host->Config.BaseAddress, ST_EMMC_FB_CLK_SEL, Status);
```

where

```
/* Core Configuration register (CORE_CONFIG) */
#define ST_EMMC_CORE_CONFIG    0x0208
/* Feedback Clock Selection register (FB_CLK_SEL) */
#define ST_EMMC_FB_CLK_SEL    0x0210
```

The Host bus clock is 100MHz (XBAR_CLK) and it’s internally provided by the microcontroller to the 3MMC; the register set and the DMA controller operate on this clock.

1.6 Signal interfaces

The interface uses several signals:

- DATA [7:0]: lines for write transaction and for data receipt during read transaction.
  - The device includes internal pull ups for all data lines.
- CMD: sends the command and receives the response coming from the external device.
- Card clock: frequency may vary according to the card speed.
- Feedback clock.

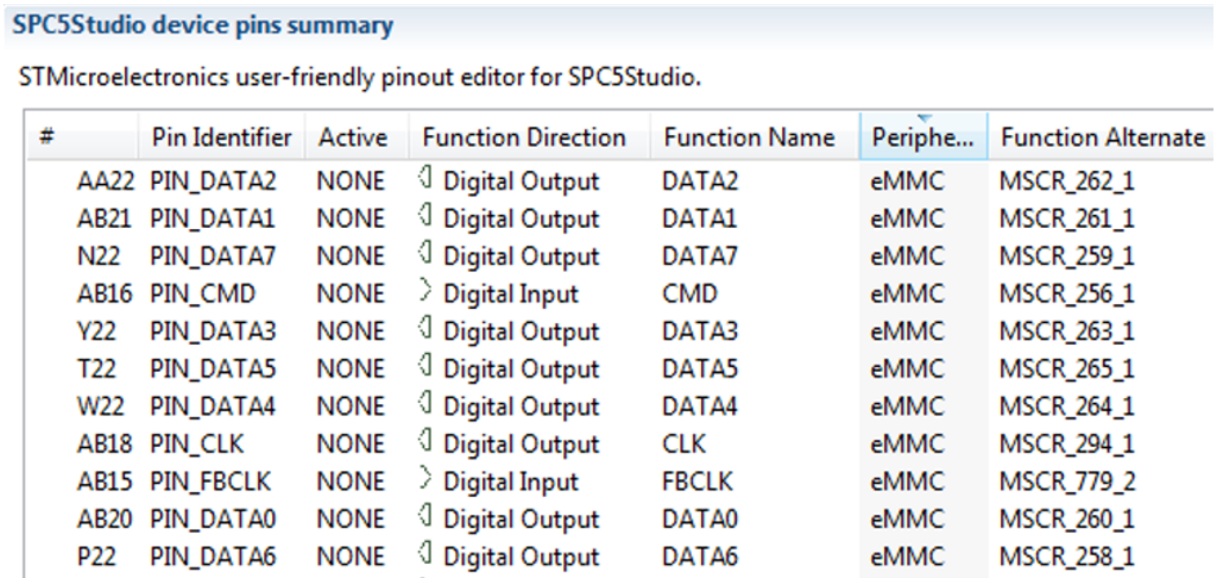
The eight data lines and the command have to be configured as digital input/output.

The feedback clock is configured as input while the card clock as bidirectional.

It is mandatory to set the Very/Ultra strong selection in the Output Edge Rate Control in the SIUL2_MSCR_IO related registers.

The pullup for data lines can be also implemented in the PAD (Weak Pullup Enable in the SIUL2_MSCR_IO related registers).

Figure 4. **SPC5Studio PIN configuration**



SPC5Studio device pins summary

STMicroelectronics user-friendly pinout editor for SPC5Studio.

#	Pin Identifier	Active	Function Direction	Function Name	Periphe...	Function Alternate
AA22	PIN_DATA2	NONE	Digital Output	DATA2	eMMC	MSCR_262_1
AB21	PIN_DATA1	NONE	Digital Output	DATA1	eMMC	MSCR_261_1
N22	PIN_DATA7	NONE	Digital Output	DATA7	eMMC	MSCR_259_1
AB16	PIN_CMD	NONE	Digital Input	CMD	eMMC	MSCR_256_1
Y22	PIN_DATA3	NONE	Digital Output	DATA3	eMMC	MSCR_263_1
T22	PIN_DATA5	NONE	Digital Output	DATA5	eMMC	MSCR_265_1
W22	PIN_DATA4	NONE	Digital Output	DATA4	eMMC	MSCR_264_1
AB18	PIN_CLK	NONE	Digital Output	CLK	eMMC	MSCR_294_1
AB15	PIN_FBCLK	NONE	Digital Input	FBCLK	eMMC	MSCR_779_2
AB20	PIN_DATA0	NONE	Digital Output	DATA0	eMMC	MSCR_260_1
P22	PIN_DATA6	NONE	Digital Output	DATA6	eMMC	MSCR_258_1

Further signals are available to manage the card insertion and writing protection. On embedded cards these are considered as optional pins.

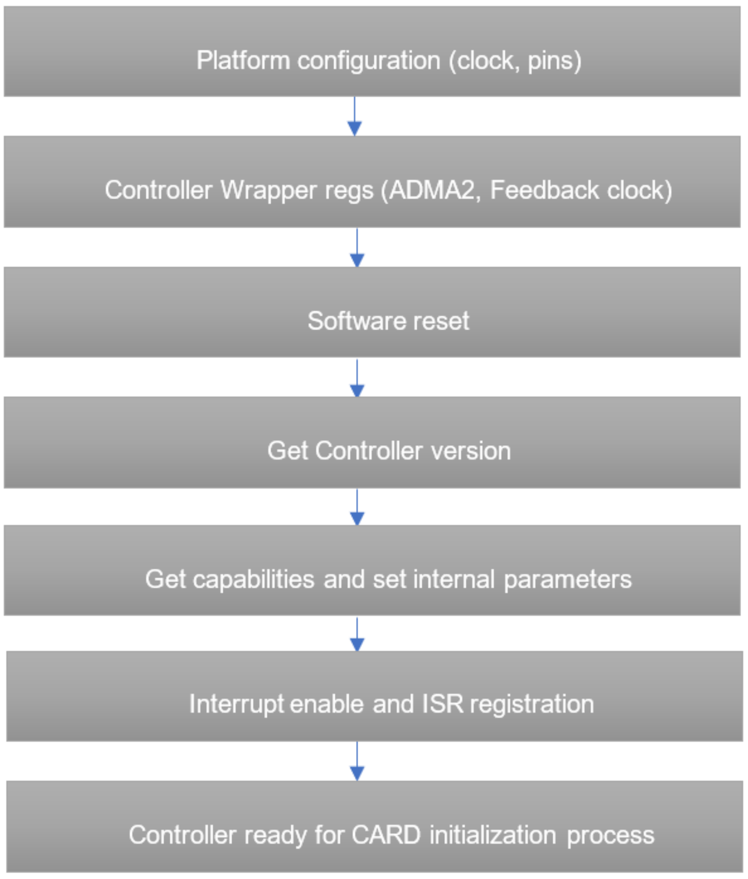
Only a set of ports of this MCU support the eMMC signals timing specification as indicated in the TN1257,*SPC58EHx, SPC58NHx IO definition: signal description and input multiplexing tables.*

For eMMC, SD and SDIO timing refer to the related specification as reported in the microcontroller's datasheet (see Section Appendix C Reference documents).

1.7 Controller initialization

Before starting the protocol sequence to detect, configure and transfer data from/to the external device, the controller must be configured by programming some internal registers.

Figure 5. **Main controller initialization steps**


Platform configuration (clock, pins)
Controller Wrapper regs (ADMA2, Feedback clock)
Software reset
Get Controller version
Get capabilities and set internal parameters
Interrupt enable and ISR registration
Controller ready for CARD initialization process

The Core configuration wrapper register provides the fields to configure the protocol clock frequency (SDMMC_CLK), as well as to select the ADMA2 and the maximum IO current value.

Before starting the configuration, a software reset can be performed by using the Software reset register.

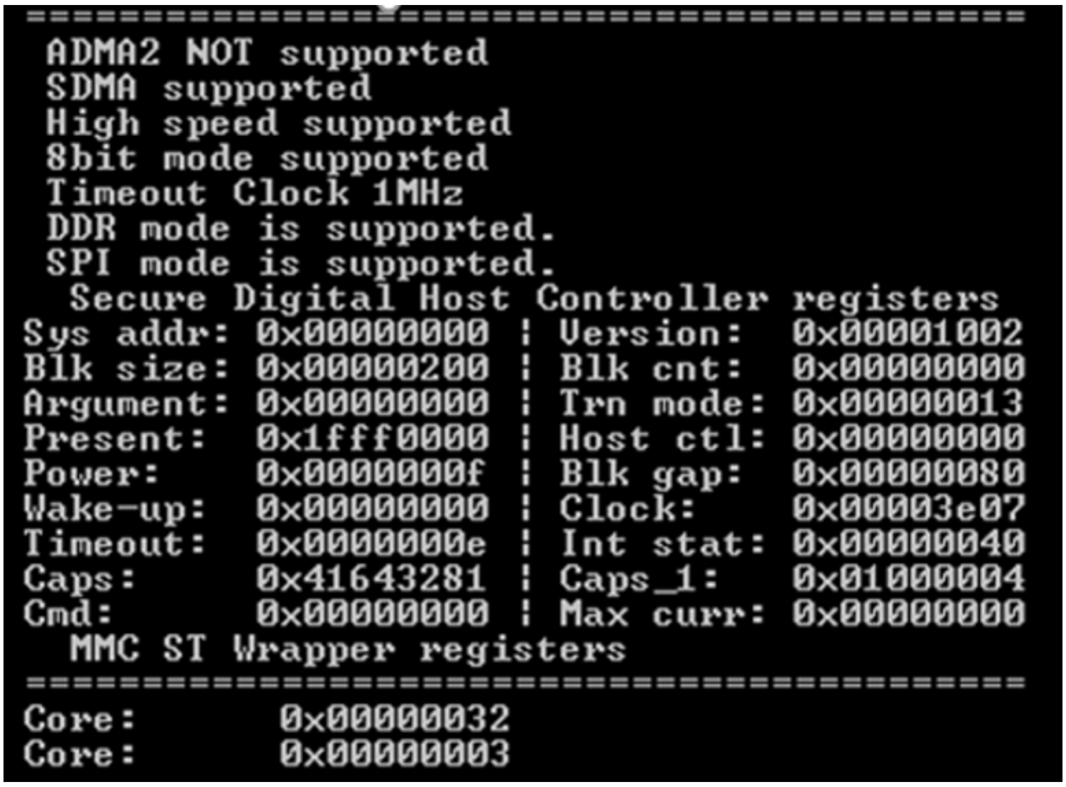
The host controller version register provides the Host specification version, which is 3.0 in this case, and the Capability registers provide the available supports embedded in this implementation. These registers can be used to tune a software driver and understand which feature will be available later configuring the card device.

A timeout control logic is available and it implements the timeout check between the block transfers. It uses the contents of the Timeout Control register. The TMCLK is used to calculate the time duration to compute timeout for the expected response from the card: its frequency is set to 1 MHz in design. By dividing it by 2^27, the timeout will be 7.45e-3 s.

The capability registers allow you to know which features are supported by the host, regarding the DMA mode, the voltage,the slot type mode (eMMC or removable card), the maximum block size, the high-speed mode and bus width.

The following figure shows an output from a serial console (used for this application) where the controller capabilities are reported.

Figure 6. Application console output



```
==========================================
 ADMA2 NOT supported
 SDMA supported
 High speed supported
 8bit mode supported
 Timeout Clock 1MHz
 DDR mode is supported.
 SPI mode is supported.
  Secure Digital Host Controller registers
Sys addr: 0x00000000 | Version:  0x00001002
Blk size: 0x00000200 | Blk cnt:  0x00000000
Argument: 0x00000000 | Trn mode: 0x00000013
Present:  0x1fff0000 | Host ctl: 0x00000000
Power:    0x0000000f | Blk gap:  0x00000080
Wake-up:  0x00000000 | Clock:    0x00003e07
Timeout:  0x0000000e | Int stat: 0x00000040
Caps:     0x41643281 | Caps_1:   0x01000004
Cmd:      0x00000000 | Max curr: 0x00000000
  MMC ST Wrapper registers
==========================================
Core:      0x00000032
Core:      0x00000003
```

1.7.1 Interrupt configuration

The following registers: Normal Interrupt status enable, Error interrupt status enable and Normal interrupt signal enable, must be configured since the beginning and the related interrupt service routine (ISR) must be installed. The interrupt source and related vector number is 820 for the 3MCR controller in this microcontroller. The interrupt service routine has to be designed to offer two services: handle normal and error events and wait for condition events. Some of the error conditions handled by ISR are Command/Data TimeOut or Command/Data CRC Error or ADMA Errors. The "wait for condition" returns when at least one of the unmasked error occurs.

Note that, while the Normal interrupt status enable register is used to mask the status for an interrupt (e.g. card insertion, transfer complete, etc.), the Normal interrupt signal enable register is actually the register used for raising the interrupt event from the hardware. For example, when implementing the polling mode in the latter register, all the fields must be masked (so reset). The same logic applies to the Error interrupt signal enable register.

1.8 Operating modes

The controller has its internal PIO/DMA module that implements the SDMA and ADMA2 engines and also provides the block transfer counts for the PIO operation. The following table summarizes the operating modes according to the host controller selected modes.

Table 3. **Operating modes (IO vs DMA)**

Modes	Card switch modes	Comments
SDMA	1,4,8 bits	• *Support 32-bit addressing format* • *No link list or descriptor table support* • *Data transfer Size is limited by the size of the block_count register(16 bit).*
ADMA2	1,4,8 bits	• *Based on a Linked List of descriptors.* • *Supports both 32b & 64b addressing.* • *No limit of data transfer size.* • *Interrupt generated at the end of data transfer corresponding to each descriptor entry.*
NON-DMA	1,4,8 bits	• *CPU usage to perform any transfer.*
SPI	1bit	• *Not supported in eMMC cards and not covered in this Application Note.*

By default the SDMA mode support is enabled. To enable the support for ADMA2, the related field must be set in the CORE_CONFIG register as shown below:

```
Status |= ST_EMMC_CORE_ADMA2_EN;
sdmmc_WriteReg32(host->Config.BaseAddress, ST_EMMC_CORE_CONFIG, Status);
```

where

```
#define ST_EMMC_CORE_ADMA2_EN (1 << 16)
```

Then the enabled support for ADMA2 can be checked in the CAPABILITIES31_0 host controller register.

1.8.1 IO mode

This mode does not use any DMA to transfer the data so the CPU usage is expected to increase. Usually this mode is mandatory for debugging purposes but it could be the only way to talk to some SDIO devices.

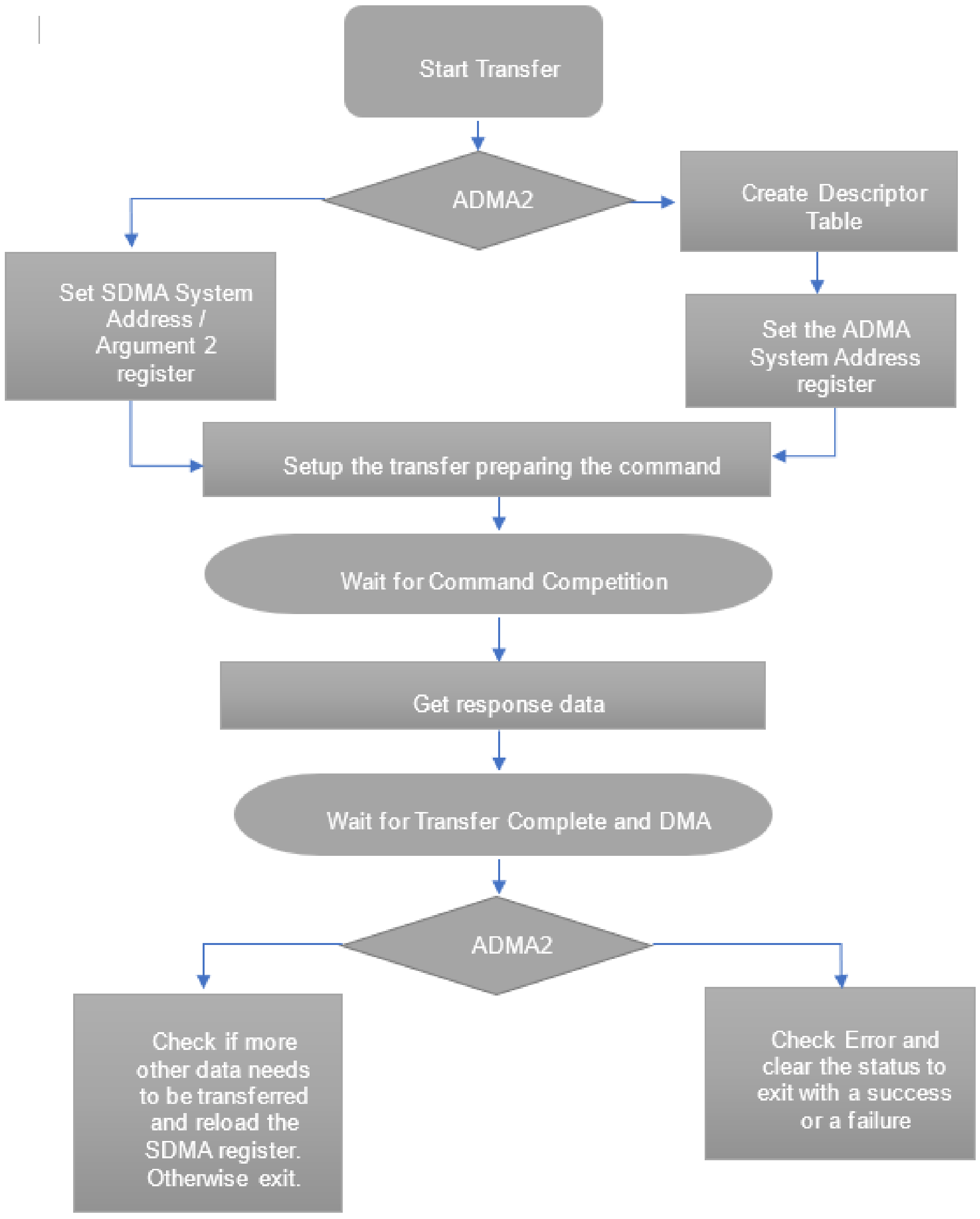
The host driver transfers data using the buffer data port register. The target bus supports only single transfer access (no burst support) and only one outstanding read/write transaction.

The data transfer using the DAT line sequence (not using DMA) and all the programming rules are fully described in the reference manual (see Section Appendix C Reference documents).

1.8.2 Single DMA vs advanced DMA2 mode

The controller has its DMA module that is able to move data from/to the external device using two different modes as already described in the previous chapters. The following figures show a simplified version of the programming flow for ADMA2 SDMA, for a detailed description of the procedure for each mode refer to the reference manual (see Section Appendix C Reference documents).

Figure 7. **Simplified version of the DMA programming flow**


Start Transfer
ADMA2
Create Descriptor Table
Set SDMA System Address / Argument 2 register
Set the ADMA System Address register
Setup the transfer preparing the command
Wait for Command Competition
Get response data
Wait for Transfer Complete and DMA
ADMA2
Check if more other data needs to be transferred and reload the SDMA register. Otherwise exit.
Check Error and clear the status to exit with a success or a failure

When using the SDMA mode, the user buffer is sent as a single chunk of data and the maximum size is fixed to the maximum block size: 65535 bytes.

The application will have to initialize the SDMA System Address / Argument 2 register before starting an SDMA transaction with the memory address (e.g. user buffer).

When the host controller stops an SDMA transfer, this register always points to the address of the next contiguous data position. After receiving the interrupt and checking that the transfer has been performed without any error, the SDMASYSADDR can be updated for the next transfer.

```
sdmmc_WriteReg32(host->Config.BaseAddress, SDMA_SYS_ADDR_OFF, User_Buff);
/* SDMA System Address Register */
#define SDMA_SYS_ADDR_OFF    0x00U
```

The DMA Controller uses its own interface to fetch ad-hoc descriptors in a linked list while operating in ADMA2 mode.

The controller can walk through a descriptor linked list without any limits on sizes, and a scatter-gather feature can be implemented in order to have multiple buffers split into a non-contiguous memory space address. Many operating systems and file system stacks can take advantage of the scatter-gather feature to improve the performance and the CPU usage.

In the ADMA mode, the application driver must set the start address of the descriptor table in the ADMA System Address register.

The ADMA increments this register address, which points to next descriptor manipulated by the engine. When an error occurs and the interrupt is generated, the ADMA System Address register will contain the latest valid descriptor address.

Before transferring in ADMA2 mode, the descriptor table has to be prepared. This is an example of the main steps that the driver has to implement:

- Get the maximum size from Block size register.
- Set the number of descriptors needed in the table (if dynamically managed).
- Fill the descriptor table: for each element the address of the data and the attribute field are set. Refer to Figure 8. ADMA2 descriptor table.
- Fill the latest descriptor closing the linked list.
- At the end, program the ADMA System Address register.

Regarding the byte swapping order, refer to the related reference manual (see Section Appendix C Reference documents) ADMA transactions chapter.

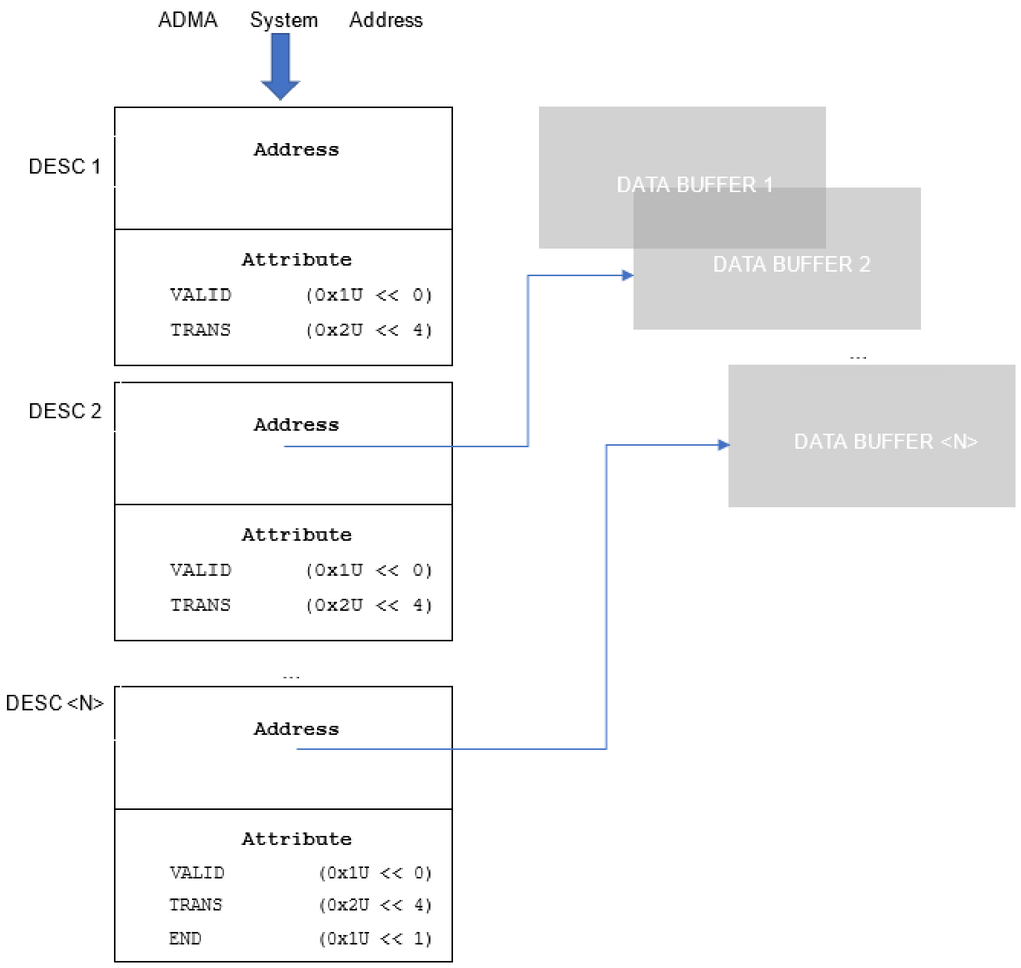


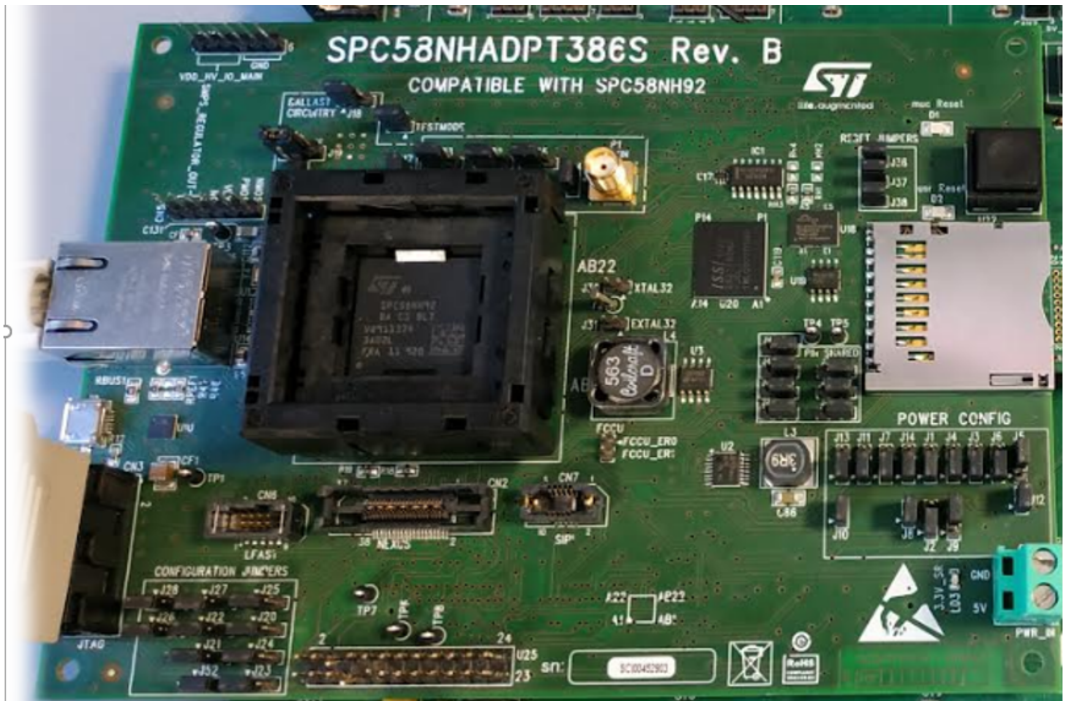
Figure 8. ADMA2 descriptor table

2 Hardware setup

The following figure shows SPC58NHADPT386S module that can be used as the reference platform to test the 3MMC controller. This module provides both the SD slot and eMMC device.

SD slot or eMMC device can be selected by switching a set of jumpers. These devices cannot be used simultaneously.

Figure 9. SPC58NHADPT386S module



The SPC58NHADPT386S embeds the IS21ES08G-JCLI eMMC device, with the following feature :

- 8-Gigabyte eMMC device
- Compliant with specification Ver.4.4, 4.41, 4.5, 5.0.
- Supports three different data bus widths : 1 bit(default), 4 bits, 8 bits
- Operating voltage range: 1.8 V, 3.3 V
- Supports Enhanced Mode, security and Field Firmware Update (FFU)
- Automotive Grade for Temperature range.

Refer to the eMMC device specification for more details about internal registers and full set of capabilities.

Figure 10. SPC58NHADPT386S jumper selection (eMMC vs SLOT)

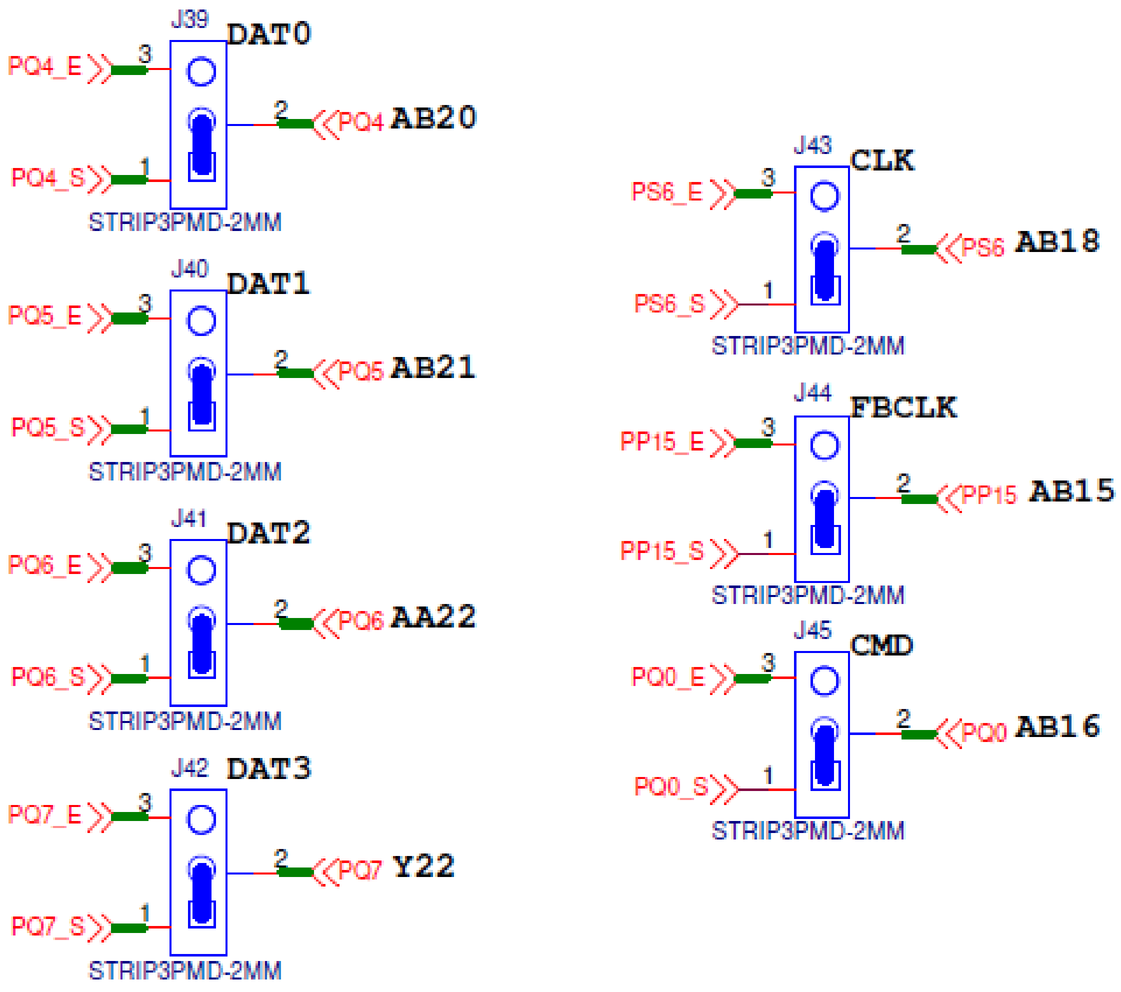
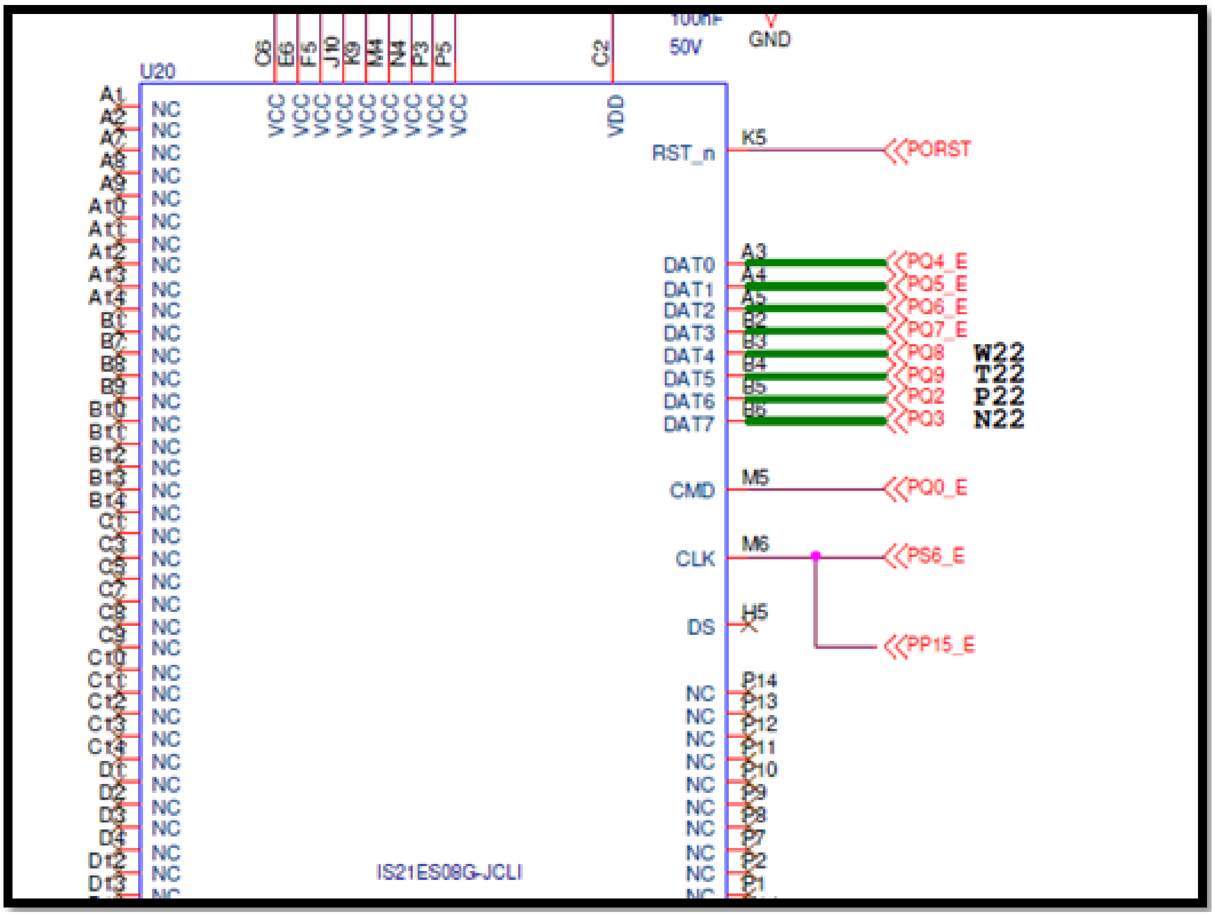


Figure 11. **IS21ES08G eMMC device connections**



The host will treat the eMMC cardas a 4.51 compatible card. In this case, some of the device features like HS400 and command queuing cannot be used.

Commercial MMC, SD devices and SDIO cards can also be tested on this hardware by using the dedicated slot, the next figure shows the connections with the microcontroller:

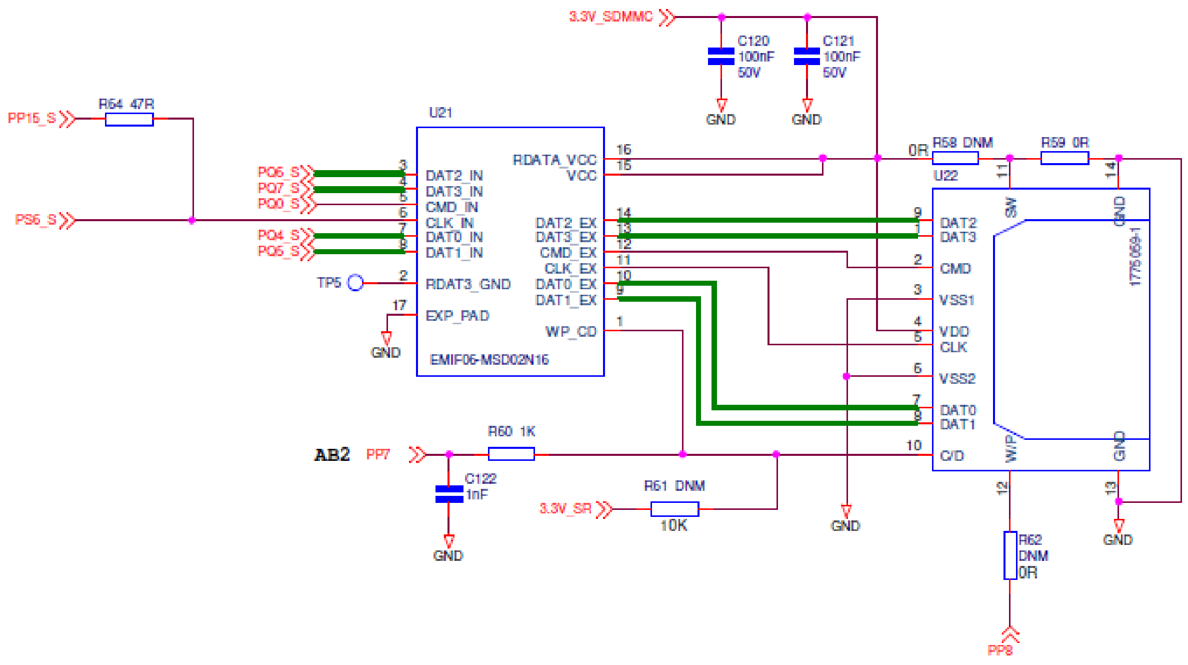


Figure 12. MMC/SD slot connections

3 Overview of the SD/MMC/SDIO specification

SD specifications part 1 "Physical Layer Specification" document version 3.01 (Feb. 2010)

After introducing the way the hardware and the controller have to be configured in this platform, and before documenting how to communicate with an external device, e.g. eMMC or SD/MMC, this chapter will provide a basic overview of the protocol, the card registers and commands.

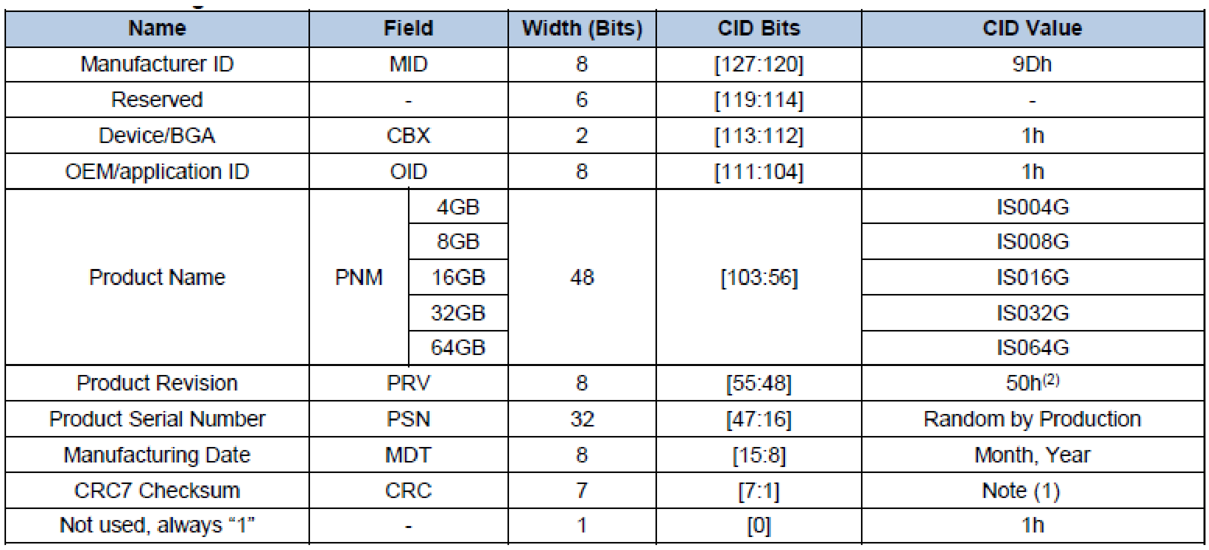
The reader should be familiar with the specification available in the standard: Embedded Multi-Media Card (eMMC), Electrical Standard (4.5 Device) JESD84-B45 (June 2011),

3.1 Card registers

Six register classes are defined within the card interface:

- The operation conditions register (OCR) stores the VDD voltage profile.
- The Card identification register (CID): Manufacturer ID, Product name, revision.
- The Card-Specific Data register (CSD) provides information on how to access the card contents. The CSD defines the data format, error correction type, maximum data access time, data transfer speed, wether the DSR register can be used, etc.
- The Extended CSD register defines the card properties and advanced/selected modes.
- The Relative card address register (RCA) carries the card address assigned by the host during the card identification.
- The Driver stage register (DSR) optionally used to improve the bus performance for extended operating conditions.

Figure 13. **IS21ES08G-JCLI CID register**

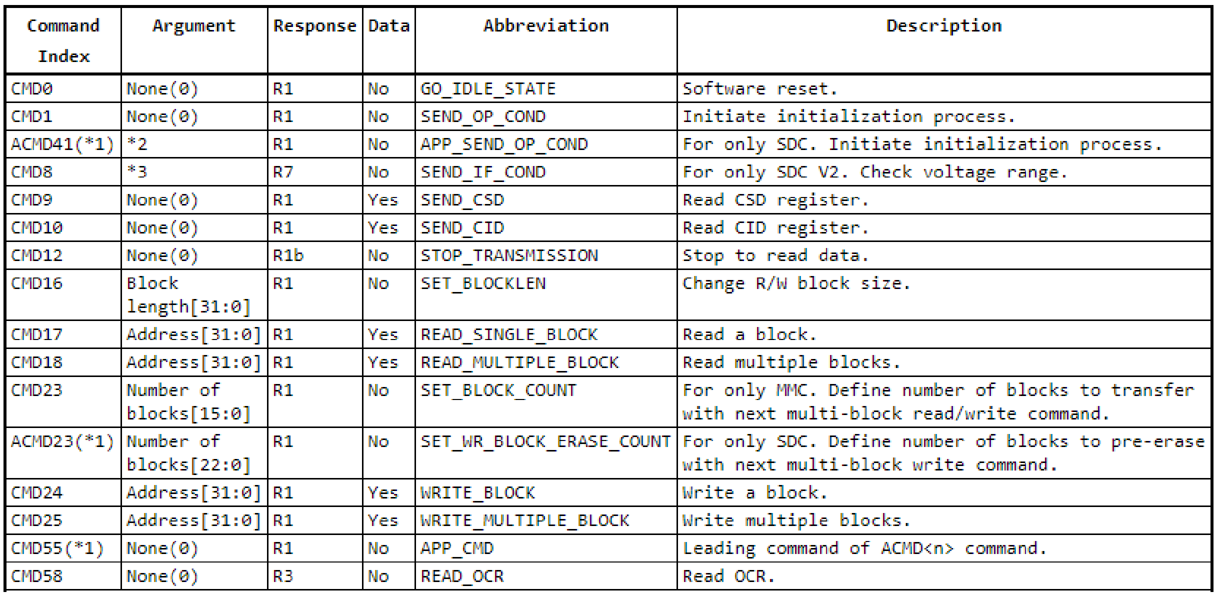


Name	Field		Width (Bits)	CID Bits	CID Value
Manufacturer ID	MID		8	[127:120]	9Dh
Reserved	-		6	[119:114]	-
Device/BGA	CBX		2	[113:112]	1h
OEM/application ID	OID		8	[111:104]	1h
Product Name	PNM	4GB	48	[103:56]	IS004G
		8GB			IS008G
		16GB			IS016G
		32GB			IS032G
		64GB			IS064G
Product Revision	PRV		8	[55:48]	50h(2)
Product Serial Number	PSN		32	[47:16]	Random by Production
Manufacturing Date	MDT		8	[15:8]	Month, Year
CRC7 Checksum	CRC		7	[7:1]	Note (1)
Not used, always “1”	-		1	[0]	1h

3.2 Card commands

Each command is expressed in abbreviations like GO_IDLE_STATE or CMD<n>, <n> is the number of the command index. A command can have parameters. The following picture shows a subset of basic commands which are usually used for generic read/write and card initialization:

Figure 14. **Basic commands index (JEDEC Standard No. 84-B45)**



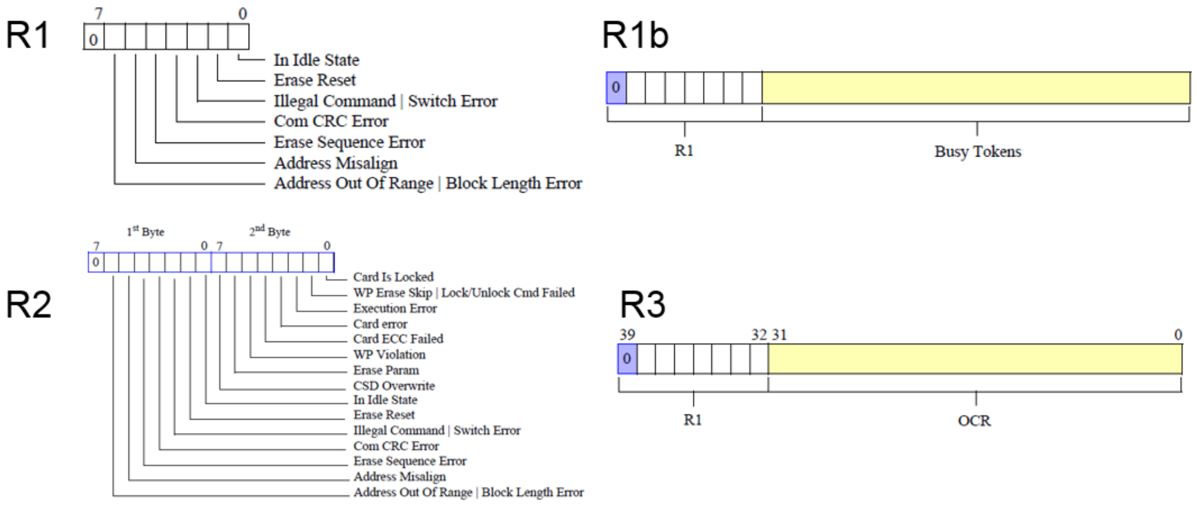
Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.

3.3 Card response

There are several types of response tokens. All responses are sent via the command CMD signal line and they are transmitted with MSB first. The commands that require a response can have different formats (as detailed in the JEDEC Standard No. 84-B45), here is a summary:

- R1 for normal response command
- R1b: R1 with an optional busy signal transmitted on the data line DATA0
- R2 for CID / CSD register
- R3 for OCR register
- R4 for Fast I/O
- R5 for Interrupt request

Figure 15. R1/R1b//R2/R3 response



3.4 Bus protocol

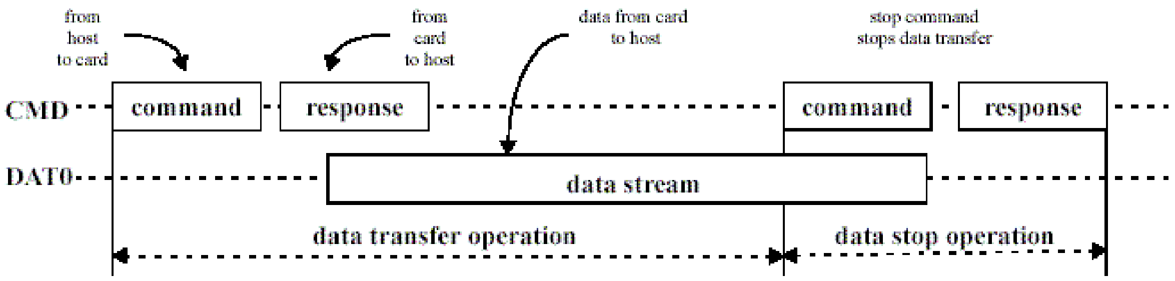
Each message is represented by command, response and data, it is useful to understand how these are addressed on the bus.

This paragraph reports, from the JEDEC Standard No. 84-B45 (Bus Protocol), the timing on sequential and multi-block operations.

3.4.1 Sequential read

This is a continuous data transfer, terminated only by a stop command, supported only in 1-bit data mode, no CRC done.

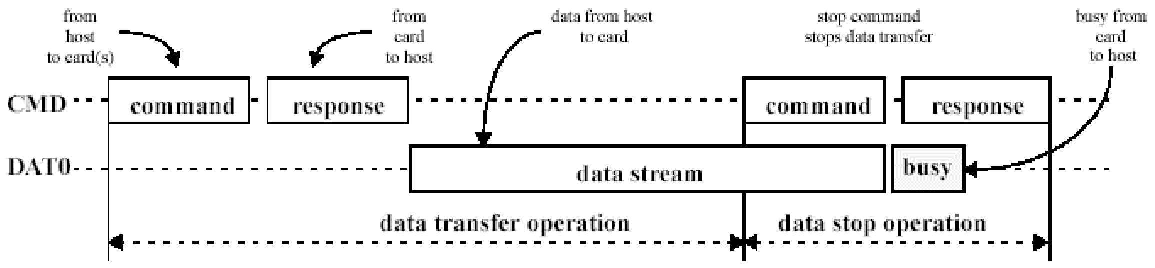
Figure 16. Sequential read operation timing



3.4.2 Sequential write

This is a continuous data transfer, terminated only by a stop command. Supported only in 1-bit data mode, no CRC done. There is a busy signaling if the card is busy.

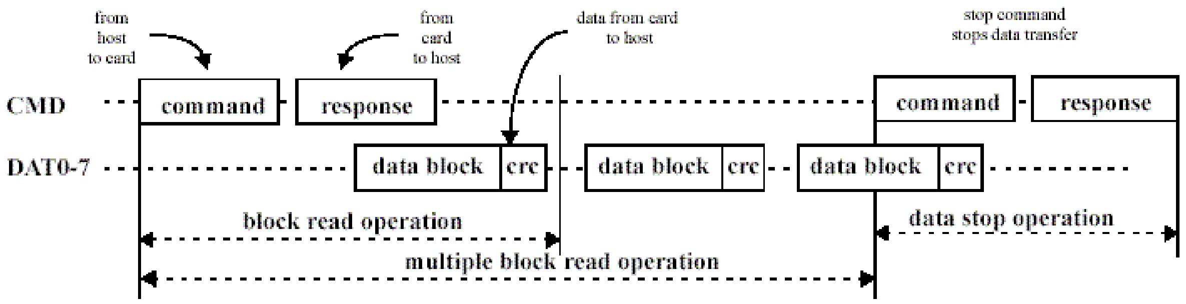
Figure 17. Sequential write operation timing



3.4.3 Multiple block read operation

In this case the data transfer requires a multiple block where the size programmable CRC is done.

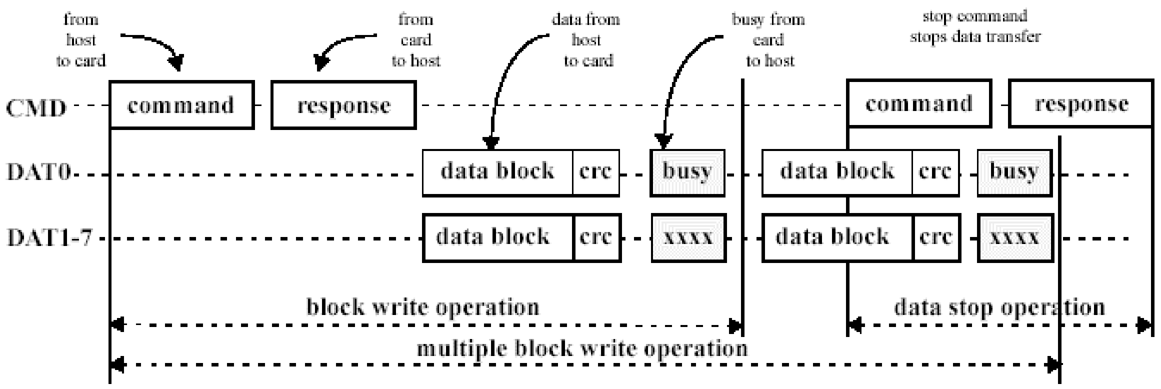
Figure 18. Multiple block read operation timing



3.4.4 Multiple block write operation

In this case the data transfer is always in multiple block, the block size is programmed in the CSD register, CRC is performed and there is a busy signaling from card by pulling DATA 0 line low.

Figure 19. Multiple block write operation timing



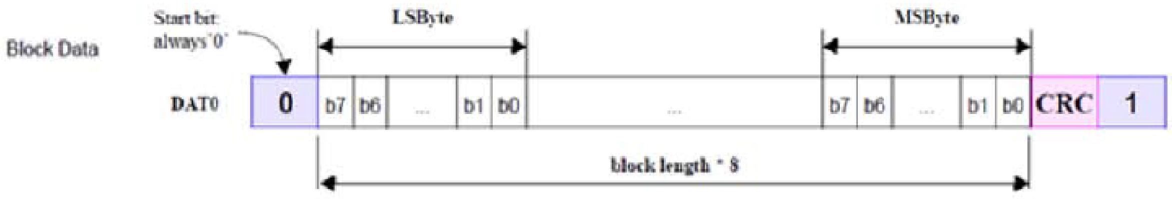
3.5 Data packet format in SDR mode

This section will explain to the reader, based on the JEDEC Standard No 84-B45, how each data is seen on the bus according to the operating width. The examples are for the SDR mode (bus sampling at rising edge).

3.5.1 1-bit bus

Only DATA0 is used in this mode:

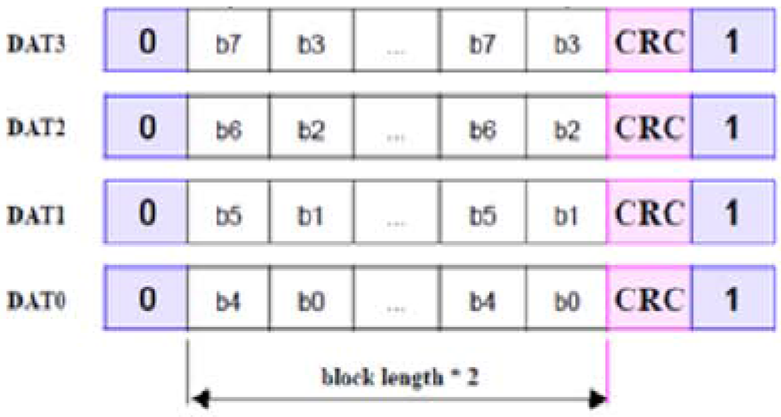
Figure 20. 1-bit bus width in SDR mode



3.5.2 4-bit bus

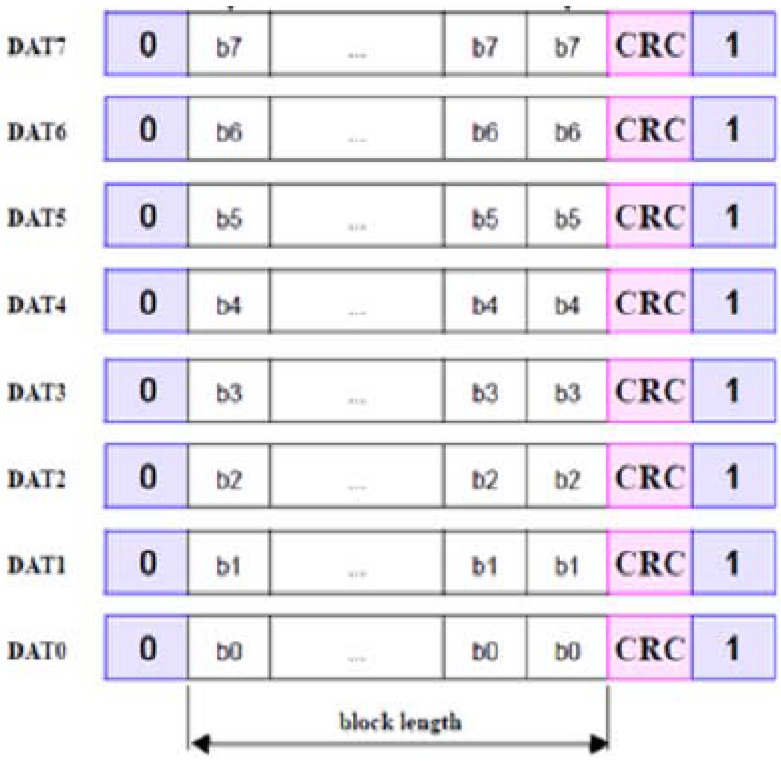
DATA[3:0] are used in this mode:

Figure 21. 4-bit bus width in SDR mode



3.5.3 8-bit bus

Figure 22. 8-bit bus width in SDR mode



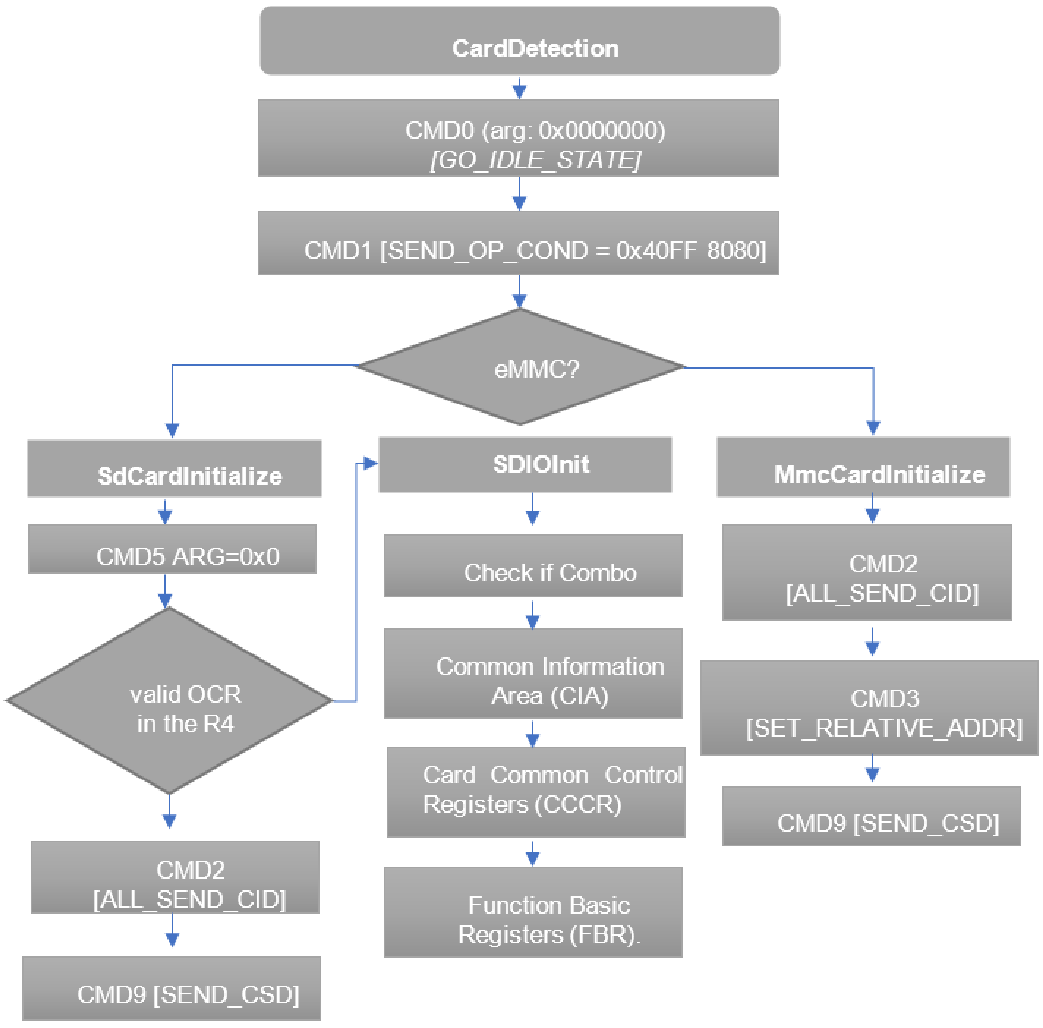
4 Card initialization

As soon as the host controller is initialized the next step is to detect the external device and configure it in the best way. The whole process is based, according to the device currently connected to the host controller, on a sequence of commands and responses, and the result is due to a sort of “agreement” between the host controller and the device.

A common approach usually followed is checking all the supported modes step-by-step, so that in case of issues the software can return to the previous configuration preventing the failure of the entire setup.

For example, the software can check if SDR25 can be configured and in case of success the SDR50 can be selected; if this mode fails, the software could decide to keep the SDR25 continuing to test the device (unless other issue occurs).

Figure 23. **Generic card detection diagram**



The generic flow diagram above aims to describe in a very simple way the main steps followed for detecting and initializing the external device.

At runtime, following the device specifications, the device type connected is detected and consequently the right command sequence is initiated for configuring it.

The controller is an SDIO-aware host so it supports the related commands (e.g. CMD5 with ARG=0x0) to detect an IO device.The reader should follow the detailed algorithm defined for SDIO-aware hosts in the SD Input/Output (SDIO) Card Specification.

After detecting the card, the controller can configure it and start moving data.

4.1 sdmmcctrl_clk frequency select

When performing the bus and high-speed switching procedures, the frequency has to be changed. At the beginning it is configured at 400 KHz, but it has to be moved to 25 MHz in case of HS selection.

On the hardware used for this application, any change to the clock can be verified on the PCB via Test Points or Jumpers (J43) with the use of an external oscilloscope. This kind of layout can help on debugging your own solutions.

The following code shows the algorithm to be used for finding and setting the divisor in the host controller clock control register according to the desired frequency. This can be used for both MMC and SD cases.

```
/* Disable clock from Control Register*/
ClockReg = sdmmc_ReadReg16(host->Config.BaseAddress, 0x2C);
ClockReg &= ~(SDMMC_CC_SD_CLK_EN_MASK | SDMMC_CC_INT_CLK_EN_MASK);
sdmmc_WriteReg16(host->Config.BaseAddress, 0x2C, ClockReg);

if (host->HC_Version == HC_SPEC_V3) {
    /* Calculate divisor */
    for (DivCnt = 0x1U; DivCnt <= SDMMC_CC_EXT_MAX_DIV_CNT; DivCnt++) {
        if (((host->Config.InputClockHz) / DivCnt) <= SelFreq) {
            Divisor = DivCnt >> 1;
            break;
}
    }
    if (DivCnt > SDMMC_CC_EXT_MAX_DIV_CNT) {
        /* No valid divisor found … exits with a failure */
…
    }
/* Set clock divisor */
    ClockReg = sdmmc_ReadReg16(host->Config.BaseAddress,    0x2C);
    ClockReg &= ~(SDMMC_CC_SDCLK_FREQ_SEL_MASK |
        SDMMC_CC_SDCLK_FREQ_SEL_EXT_MASK);

    ExtDivisor = Divisor >> 8;
    ExtDivisor <<= SDMMC_CC_EXT_DIV_SHIFT;
    ExtDivisor &= SDMMC_CC_SDCLK_FREQ_SEL_EXT_MASK;
    Divisor <<= SDMMC_CC_DIV_SHIFT;
    Divisor &= SDMMC_CC_SDCLK_FREQ_SEL_MASK;
    ClockReg |= Divisor | ExtDivisor |(uint16_t)SDMMC_CC_INT_CLK_EN_MASK;
    sdmmc_WriteReg16(host->Config.BaseAddress, 0x2C, ClockReg);
}

/* Wait for internal clock to stabilize */
ReadReg = sdmmc_ReadReg16(host->Config.BaseAddress, 0x2C);
while((ReadReg & SDMMC_CC_INT_CLK_STABLE_MASK) == 0U) {
    ReadReg = sdmmc_ReadReg16(host->Config.BaseAddress, 0x2C);;
}
/* Enable clock again */
ClockReg = sdmmc_ReadReg16(host->Config.BaseAddress, 0x2C);
sdmmc_WriteReg16(host->Config.BaseAddress, 0x2C,
ClockReg | SDMMC_CC_SD_CLK_EN_MASK);
Where:
#define SDMMC_CC_INT_CLK_EN_MASK        0x1U
#define SDMMC_CC_INT_CLK_STABLE_MASK    0x2U
#define SDMMC_CC_SD_CLK_EN_MASK        0x4U
```

4.2 SD configuration

This is a summary of the main steps after the SD card is detected, i.e. the software driver has to check the card version, speed and start the voltage sequence. To read the SD Configuration Register (SCR), the ACMD51 has to be sent. The R0 will contain the SCR that can be saved in a private structure.

For a detailed description of the entire SD card power up and configuration, refer to SD specifications, Part 1, Physical layer"

```
scr_reg = (host->SCR[0] & 0xf0) >> 4;
/* SD_SPEC
 * 0 Version 1.0 and 1.01
 * 1 Version 1.10
 * 2 Version 2.00 or Version 3.00 (Refer to SD_SPEC3)
 */
host->card.scr.sda_spec = host->SCR[0] & 0xf;
host->card.scr.sec = (host->SCR[1] & 0x70) >> 4;
/* DAT Bus widths supported
 * Bit 0 1 bit (DAT0)
 * Bit 2 4 bit (DAT0-3)
 */
host->card.scr.bus_widths = host->SCR[1] & 0xf;
/* Spec. Version 3.00 or high */
host->card.scr.sda_spec3 = (host->SCR[1] & 0x80) >> 7;
```

If the 4-bit mode is supported the software can try to change the bus mode issuing the ACMD6.

The bus width in the Host Controller registers 1 has to be configured according to the mode available.

The maximum frequency and the maximum current are determined by CMD6, so the software can issue the command with the appropriate argument below to get and set the speed:

```
#define SDMMC_SWITCH_CMD_HS_GET    0x00FFFFF0U

/* access mode [3:0] for SDR12…SDR50 */
#define SDMMC_SWITCH_CMD_SDR12_SET    0x80FFFFF0U
#define SDMMC_SWITCH_CMD_SDR25_SET    0x80FFFFF1U
#define SDMMC_SWITCH_CMD_SDR50_SET    0x80FFFFF2U
```

Note that, while getting the speed, this is saved in R0.

4.3 MMC configuration

In case of an eMMC device, the bus switch can be 8, so the software validates and configures it if supported.

Sending the CMD8 SEND_EXT_CSD the extended registers will be retrieved from the device. As documented in the JEDEC Standard No. 84-B45, these registers define the device properties and selected modes. The software will use them at runtime, for example to change the bus width (BUS_WIDTH [183]).

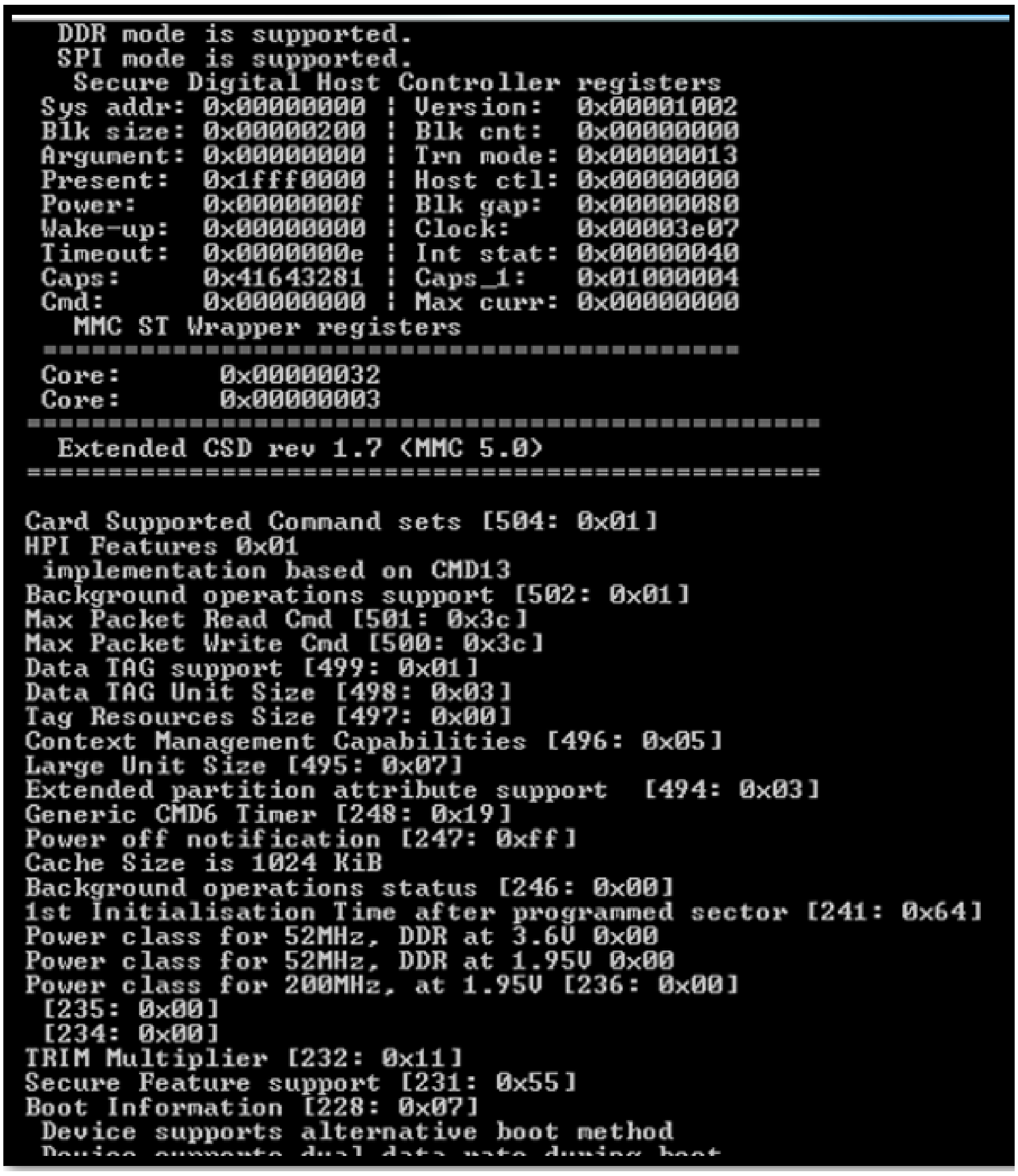
By checking the DEVICE_TYPE [196] you can verify if the device is a high-speed dual data rate or high-speed at 50 or 25 MHz.

Every time the software performs any switch the EXT_CSD can be analyzed to understand if the desired configuration is adopted.

While switching to DDR50 mode, the software must enable the Feedback clock and the mode in the Host control 2 register as shown below:

```
/* Enable DDR in the host: hostctrl2_uhsmodeselect = 0b100 */
RegValue = (sdmmc_ReadReg8(host->Config.BaseAddress, 0x3e)) |= 0x4;;
sdmmc_WriteReg8(host->Config.BaseAddress, 0x3e, RegValue);
```

Figure 24. eMMC card - Extended CSD dump



```
   DDR mode is supported.
   SPI mode is supported.
    Secure Digital Host Controller registers
  Sys addr: 0x00000000 | Version:  0x00001002
  Blk size: 0x00000200 | Blk cnt:  0x00000000
  Argument: 0x00000000 | Trn mode: 0x00000013
  Present:  0x1fff0000 | Host ctl: 0x00000000
  Power:    0x0000000f | Blk gap:  0x00000080
  Wake-up:  0x00000000 | Clock:    0x00003e07
  Timeout:  0x0000000e | Int stat: 0x00000040
  Caps:     0x41643281 | Caps_1:   0x01000004
  Cmd:      0x00000000 | Max curr: 0x00000000
    MMC ST Wrapper registers
  ============================================
  Core:      0x00000032
  Core:      0x00000003
=================================================
   Extended CSD rev 1.7 (MMC 5.0)
=================================================

Card Supported Command sets [504: 0x01]
HPI Features 0x01
 implementation based on CMD13
Background operations support [502: 0x01]
Max Packet Read Cmd [501: 0x3c]
Max Packet Write Cmd [500: 0x3c]
Data TAG support [499: 0x01]
Data TAG Unit Size [498: 0x03]
Tag Resources Size [497: 0x00]
Context Management Capabilities [496: 0x05]
Large Unit Size [495: 0x07]
Extended partition attribute support  [494: 0x03]
Generic CMD6 Timer [248: 0x19]
Power off notification [247: 0xff]
Cache Size is 1024 KiB
Background operations status [246: 0x00]
1st Initialisation Time after programmed sector [241: 0x64]
Power class for 52MHz, DDR at 3.6V 0x00
Power class for 52MHz, DDR at 1.95V 0x00
Power class for 200MHz, at 1.95V [236: 0x00]
 [235: 0x00]
 [234: 0x00]
TRIM Multiplier [232: 0x11]
Secure Feature support [231: 0x55]
Boot Information [228: 0x07]
 Device supports alternative boot method
 Device supports dual data rate during boot
```

5 Read/Write (Single/Multi block)

The following APIs show a very simple example of how to perform a single- or multi-block transfer in polling mode. The upper layer should be aware of the block count and pass the buffer where data has to be transferred on.

```
int32_t sdmmc_Read(sdmmc *host, uint32_t Arg, uint32_t BlkCnt,uint8_t *Buf)
{
…
    if (BlkCnt > 1) {
        sdmmc_SetupXfer(host, BlkCnt, SDMMC_BLK_SIZE_512_MASK, Buf,
                    SD_MMC_READ_XFER, SDMMC_TM_AUTO_CMD12_EN_MASK);
        Status = sdmmc_CmdTransfer(host, CMD18, Arg);
        /* check the status and fails in case of errors */
    } else {
        sdmmc_SetupXfer(host, BlkCnt, SDMMC_BLK_SIZE_512_MASK, Buf,
    SD_MMC_READ_XFER, 0);
        Status = sdmmc_CmdTransfer(host, CMD17, Arg);
        /* check the status and fails in case of errors */
    }

    /* Check for transfer complete (polling mode), failing on errors */
    Status = sdmmc_CheckXferStatus(host);
    if (Status != pdTRUE) {
        return pdFALSE;
    }
    /* Check for transfer
    sdmmc_ReadReg32(host->Config.BaseAddress, SDMMC_RESP0_OFFSET);

    return pdTRUE;
}

int32_t sdmmc_Write(sdmmc *host,uint32_t Arg,uint32_t BlkCnt, uint8_t *Buf)
{
…
    /* Send block write command for single or multiple transfer */
    if (BlkCnt > 1) {
sdmmc_SetupXfer(host, BlkCnt, SDMMC_BLK_SIZE_512_MASK, Buf,
SD_MMC_WRITE_XFER, SDMMC_TM_AUTO_CMD12_EN_MASK);
        Status = sdmmc_CmdTransfer(host, CMD25, Arg);
/* check the status and fails in case of errors */
    } else {
        sdmmc_SetupXfer(host, BlkCnt, SDMMC_BLK_SIZE_512_MASK, Buf,
    SD_MMC_WRITE_XFER, 0);
        Status = sdmmc_CmdTransfer(host, CMD24, Arg);
/* check the status and fails in case of errors */
        }
    }
    sdmmc_CheckXferStatus(host);

    return Status;
}
```

The `sdmmc_SetupXfer` function is called to setup the data to be transferred from/to the card. It programs the Block Size and Block Size host controller registers and, in case of ADMA2, prepares the descriptor table. Then it sets the direction in the Transfer mode register. The direction means: read or write operation.

Multiple-block read and write commands for memory require CMD12 to stop the operation.

In this case, the Host controller issues the CMD12 automatically when last block transfer is completed. Auto CMD12 error is indicated to the Auto CMD Error Status register.

The `sdmmc_CmdTransfer` function will be called to issue the following commands to write data:

- *CMD24 (WRITE_BLOCK)*
- *CMD25 (WRITE_MULTIPLE_BLOCK)*

While the following commands to read on DATA line(s):

- *CMD17 (READ_SINGLE_BLOCK)*
- *CMD18 (READ_MULTIPLE_BLOCK)*

The `sdmmc_CmdTransfer` function is designed to generate commands to the card, here below is a simple flow diagram:

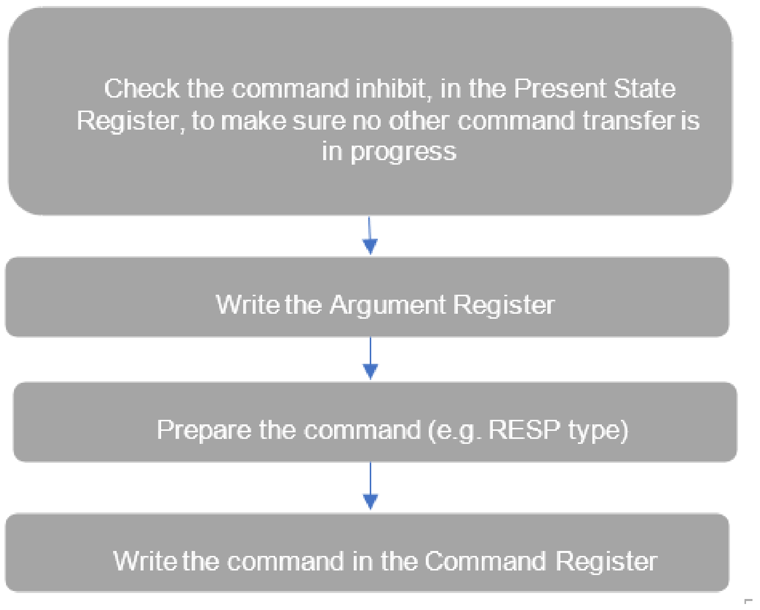

Check the command inhibit, in the Present State Register, to make sure no other command transfer is in progress
Write the Argument Register
Prepare the command (e.g. RESP type)
Write the command in the Command Register

Figure 25. **Command transfer diagram**

If polling mode is used, the routine can directly check the status inside the Normal and Status registers and clear the fields reporting error or success conditions.

6 Interrupt service routine

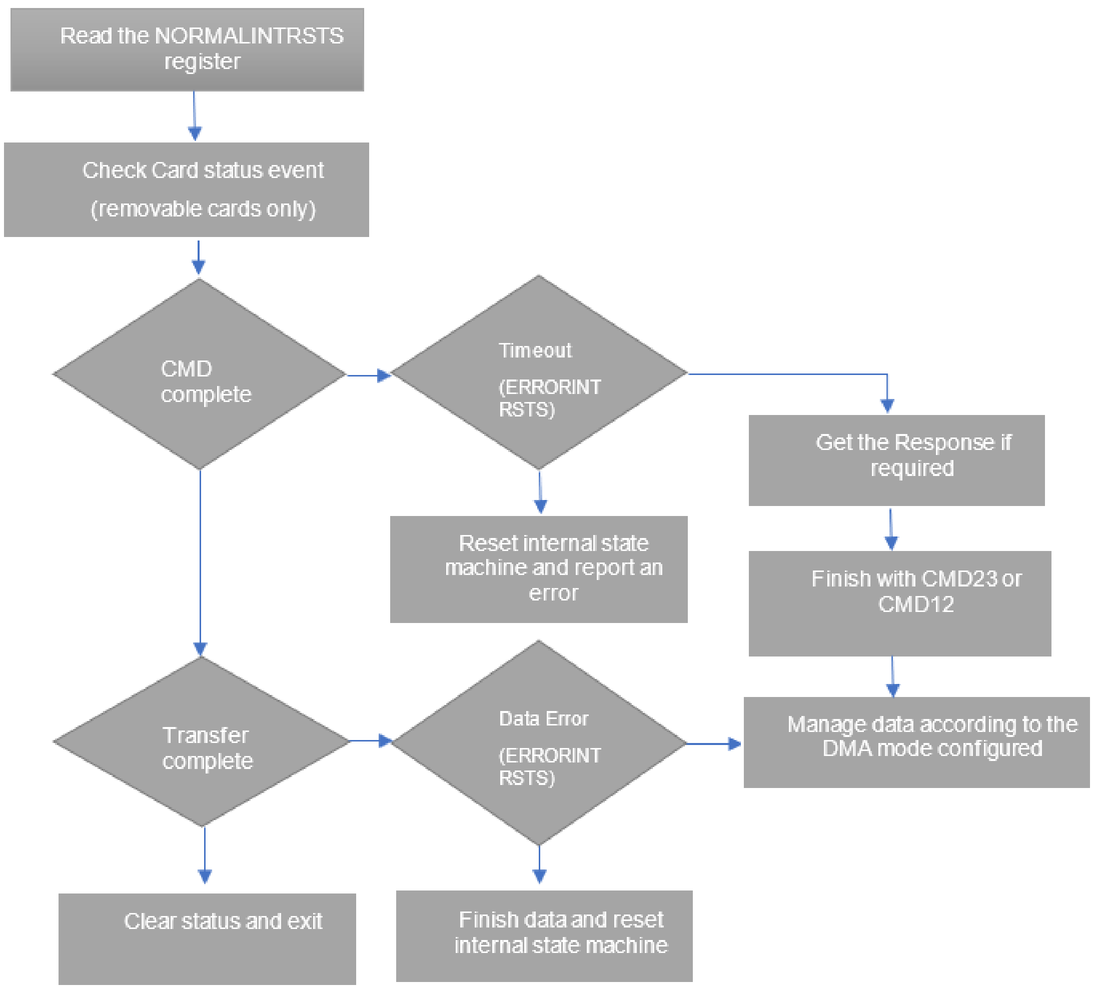
The interrupt service routine is invoked by the controller to handle different types of events (NORMALINTRSTS), for example:

- Read/write transaction is completed (NORMALINTRSTS_XFERCOMPLETE).
- Command complete (NORMALINTRSTS_ CMDCOMPLETE), which is set when we get the end bit of the command response.
- Block gap event (NORMALINTRSTS_BLKGAPEVENT).
- DMA interrupt (NORMALINTRSTS_DMAINTERRUPT).
- Card removal/Insertion and card interrupt (e.g. SDIO).

If any of the bits in the Error Interrupt Status register are set, then REG_ERRORINTRSTS is set and the ISR can check the Error interrupt status register.

The simple flow diagram below shows a typical implementation of the ISR for this controller:

Figure 26. **ISR: handling CMD and data events**


Read the NORMALINTRSTS register
Check Card status event
(removable cards only)
CMD complete
Timeout
(ERRORINT RSTS)
Get the Response if required
Reset internal state machine and report an error
Finish with CMD23 or CMD12
Transfer complete
Data Error
(ERRORINT RSTS)
Manage data according to the DMA mode configured
Clear status and exit
Finish data and reset internal state machine

7 Conclusions

The eMMC/SD memory allows users to meet the needs of high-capacity storage and high-reliability applications in automotive microcontrollers. These devices can be considered a very good choice for Flash storage in automotive context. Many eMMC devices in the market are already targeted as automotive-grade products allowing the final customer to have a large choice between different devices with different speed and cost price. These high-capacity cards can be used for several purposes, for infotainment applications, but also for FOTA that can be easily implemented (more firmware files can be stored according to the desired profiles).

The SPC58EHx/SPC58NHx automotive microcontroller also provides reference board designs where both eMMC device and SD/MMC/SDIO cards can be tested covering all the user needs. In case of IO cards the application does not enter in details, if IO_SEND_OP_COND (CMD5) returns a valid OCR, a complex software driver should be implemented to talk to the IO part in the card and in ad-hoc driver could be useful to test it (e.g. WiFi or Bluetooth devices). The eMMC devices have many features as showed in the JEDEC Standard No. 84-B45 specification: hardware partitions, different speed and bus modes to cover several different needs. A common practice is to have a file system on a card (either SD or MMC or SDIO combo that combines IO device plus memory). The selection of a FS is out of the scope of this application such as the operating system that can run on the micro controller.

This application note provides a getting started tutorial to understand how the controller and the card work and how to implement a low-level driver.

The purpose of the document is to explain the key parts of the supported standards, which are repeatedly mentioned across the document, keeping the focus on the complete guides that must be studied before.

Having clear how to develop the main set of low-level APIs, an operating system (available on the final application) should just schedule a task to initialize the controller, detect and configure the card as described in the previous chapters. The Read/Write functions implemented in a FS stack should just invoke the low-level APIs to transfer single/multiple blocks.

According to the OS and FS (and their related configuration), the overall performance of the controller can change. This application will not show any benchmark figures when use a File System.

The expected performance has been achieved, that is around 800 Mbits (100 MBytes) per second data rate using 8-bit parallel data lines (DDR50 mode), raw device accesses.

Appendix A Host controller

The Host controller has a subset of registers with different address size so the following define commands have been provided:

```
#define REG_WRITE8(address, value)        \
((*(volatile uint8_t*)(address)) = (uint8_t)(value))
#define REG_WRITE16(address, value)       \
((*(volatile uint16_t*)(address)) = (uint16_t)(value))
#define REG_WRITE32(address, value)       \
((*(volatile uint32_t*)(address)) = (uint32_t)(value))
#define REG_READ8(address)                (*(volatile uint8_t*)(address))
#define REG_READ16(address)               (*(volatile uint16_t*)(address))
#define REG_READ32(address)               (*(volatile uint32_t*)(address))
#define sdmmc_ReadReg32(BaseAddress, RegOffset)              \
        REG_READ32((BaseAddress) + (RegOffset))
#define sdmmc_WriteReg32(BaseAddress, RegOffset, RegisterValue)     \
        REG_WRITE32((BaseAddress) + (RegOffset), (RegisterValue))
#define sdmmc_ReadReg16(BaseAddress, RegOffset)              \
        REG_READ16((BaseAddress) + (RegOffset))
#define sdmmc_WriteReg16(BaseAddress, RegOffset, RegisterValue)     \
        REG_WRITE16((BaseAddress) + (RegOffset), (RegisterValue))
#define sdmmc_ReadReg8(BaseAddress, RegOffset)                 \
        REG_READ8((BaseAddress) + (RegOffset))
#define sdmmc_WriteReg8(BaseAddress, RegOffset, RegisterValue)     \
        REG_WRITE8((BaseAddress) + (RegOffset), (RegisterValue))
```

Appendix B Acronyms and abbreviations

Table 4. Acronyms

Abbreviation	Complete name
SDMA	Single Direct memory address
ADMA	Advanced DMA
MMC	Multi Media Card
eMMC	Embedded MMC
SD	Secure Digital
SDIO	Secure Digital Input Output
OS	Operating System
FS	File system
UHS	Ultra-High Speed
FOTA	Firmware over the Air

Appendix C Reference documents

- RM0452 Reference manual
- SPC58EHx, SPC58NHx datasheet
- TN1257 SPC58EHx, SPC58NHx IO definition: signal description and input multiplexing tables
- IS21/22ES04G/08G/16G/32G/64G datasheet
- SPC58EHx, SPC58NHx Errata sheet
- EMBEDDED MULTI-MEDIA CARD (eMMC), ELECTRICAL STANDARD (4.5 Device) JESD84-B45 (June 2011)
- SD Specifications Part 1 Physical Layer Specification" document version 3.01 (Feb. 2010)
- SD Input/Output (SDIO) Card Specification
- PartA2 SD Host Controller Simplified Specification Ver3.00.

Revision history

Table 5. **Document revision history**

Date	Version	Changes
19-Nov-2020	1	Initial release.

Contents

List of tables

List of figures