**SD/eMMC/MMC Card hardware testing unit**

# MMC Test Pack : synthesizable cores
# Technical Reference Manual

Written by:

**John Clayton**
Klugwhallah FPGA design team

March 17, 2017

## Table of Contents

# 1 List of Acronymns

| | |
|---|---|
| DDR | Double Data Rate |
| FPGA | Field Programmable Gate Array |
| JTAG | Joint Test Access Group |
| MMC | Multi-Media Card |
| SD | Secure Digital |
| USART | Universal Synchronous/Asynchronous Receiver/Transmitter |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |

# 2 Introduction

This document provides a description of the interface signals, internal structure and registers present in the synthesizable cores within the VHDL package file named "mmc_test_pack.vhd" Some of the cores may be intended for use at lower levels of a design, in hierarchical fashion. All cores in the VHDL package are synthesizable and have been tested via simulation and in hardware using a

Xilinx “ARTY” Artix 7 FPGA development board.

# 3 Description of Cores

## 3.1 Background

The “mmc_test_pack.vhd” VHDL package consists of two VHDL entities which combine with other cores from “sd_card_pack.vhd,” “sd_host_pack.vhd” and nine other VHDL packages, to realize a core that facilitates simulation, hardware testing and data logging of traffic on a Multi-Media Card (MMC) or Secure Digital (SD) card bus. The main specification document used during development was the JEDEC document JESD84-A44 titled “Embedded MultiMediaCard(e•MMC)” (MMCA, 4.4) version.

## 3.2 Summary of Cores In Package

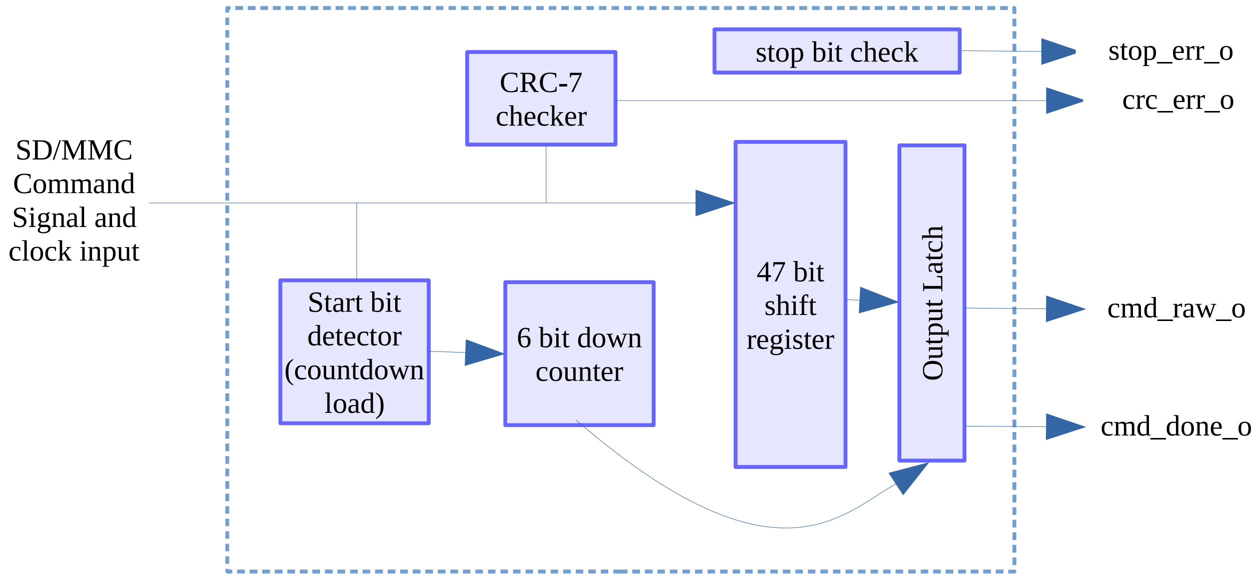
The cores present in the package file “mmc_test_pack.vhd” are shown in table Table 1. The last core in the table was written specifically for the task at hand, namely to provide an FPGA based data logger and test platform for working with SD/MMC hosts and cards. The functional capability of the mmc_tester core is such that it can be used in several different ways.

| Name | Description |
|---|---|
| mmc_test_cmd_rx | Card command receiver core, testing version |
| mmc_tester | Card and host testing, card datalogging core |

*Table 1: MMC Test Pack Cores*

## 3.3 mmc_test_cmd_rx

The first core is one of the most elementary and basic, a special testing version of a SD/MMC command receiver. Its internal structure is shown in this block diagram:



The mmc_test_cmd_rx captures all traffic going across the bus in either direction, both commands and responses, and it attempts to correctly capture the first 48 bits of the communication. This was done as a trade off, so that all traffic could be captured in 48 bit segments, and longer 136 bit messages are simply truncated in the interest of simplicity. Just to reiterate, the mmc_test_cmd_rx captures traffic from both directions, and has logic to correctly ignore the “tail end” of longer messages, and in these ways it differs from the “regular” sd_card_cmd_rx receiver in sd_card_pack.vhd.

This module is meant to be part of a system that tests, or snoops on an SD/MMC card.

This module clocks incoming serial bits from the cmd signal into a 48 bit shift register. It starts when a '0' (start) bit is found, and then shifts in 47 additional bits. The expected format of the command is:

0 d [index] [arg] [crc] 1

Where:

| | |
|---|---|
| d | = direction bit. '1'=host to card, '0'=reply from card. |
| index | = 6 bits |
| arg | = 32 bits |
| crc | = 7 bits |

The test command receiver checks that the last bit is a '1' (stop) bit, in the expected position (bit 47; start bit is bit 0). The seven bits immediately prior to the stop bit are checked using a CRC-7 code.

If any of the checks do not pass, the associated error bits are set. Regardless of errors, however, the received data bits are presented at the cmd_raw_o output. The cmd_rx_done_o output is pulsed high for one clock cycle to notify the downstream entities that newly received data is available.

In the interest of simplicity and in an effort to remain practical, only 48 bit responses are captured correctly by this unit. For R2 responses, which are 136 bits long, only the first 48 bits are presented.

Because the R2 type responses are 136 bits long, specific logic is included to try and detect them, and prevent errors from being flagged by their occurrence.  The first eight bits of an R2 response are "00111111".  Unfortunately, the first eight bits of a 48 bit R3 response are also "00111111". Fortunately, the CRC field of an R3 response is always set to "1111111".  So, the logic checks for these conditions and behaves accordingly.  In the case of an R2 response, that includes ignoring further activity on the command line for another 88 clocks after the initial 48 bits are received.

Note that this receiver runs entirely within the sd_clk_i clock domain.  Therefore, care must be taken when using the outputs.  A FIFO can form a natural "clock domain boundary crossing" or the user may need to implement other special handshaking to safely transfer signals into a different clock domain.

## 3.4 mmc_tester

Now, we get into the real “meat” of this VHDL package.

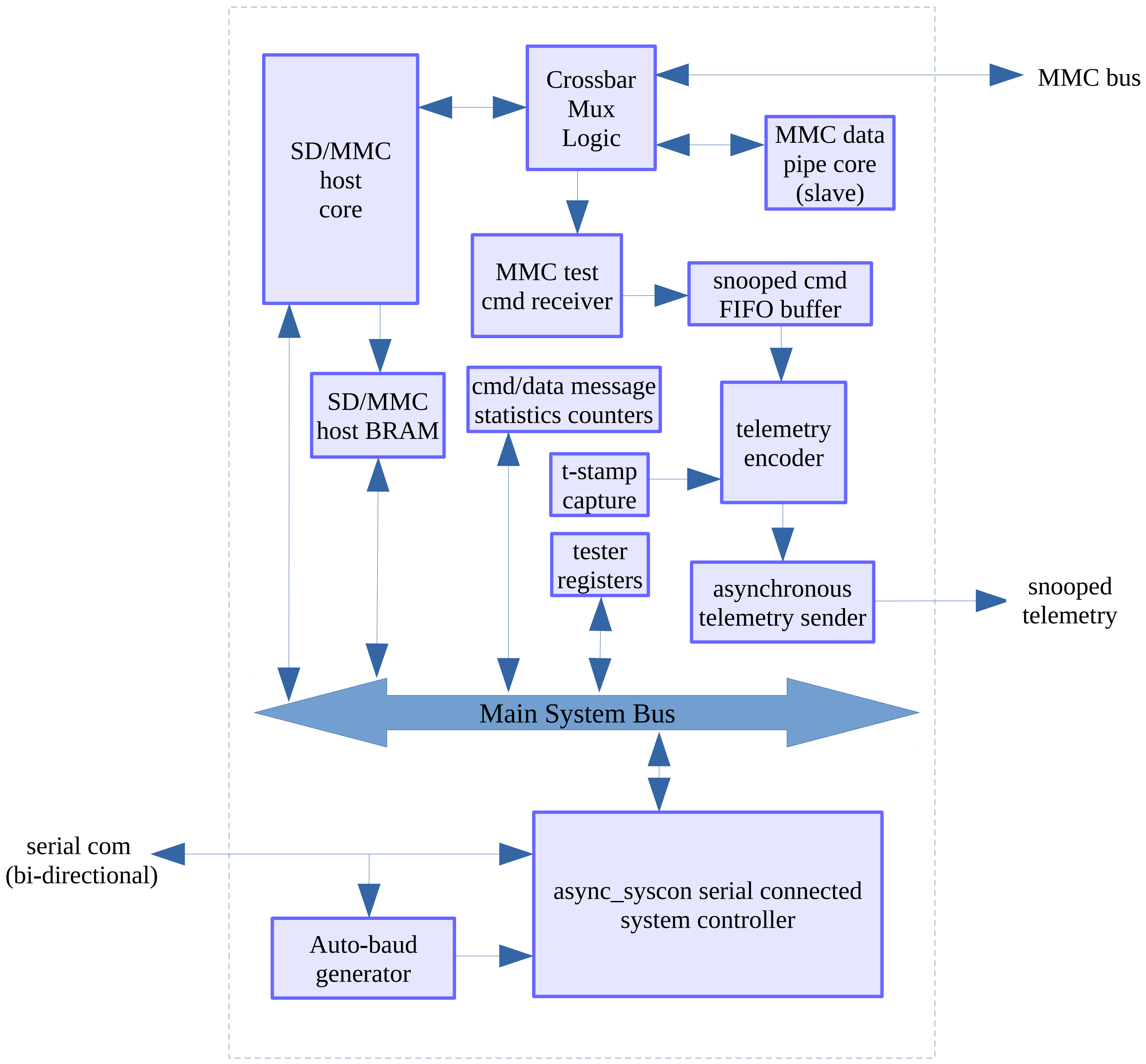
This core, mmc_tester, is a synthesizable top level unit which brings together many useful pieces in order to allow testing of devices which share an MMC cardbus. It also allows snooping of the traffic over that bus, providing a means for sending out the resulting timestamped “telemetry” for archiving. To approach the structure of this core in an analytical way, it will help to state, at the outset, that the lower level units which compose the mmc_tester comprise five main functions:

1. An sd_controller_8bit_bram core as an MMC host, with some BRAM connected to give the host some storage area, plus some registers to allow working with the host.
2. An mmc_data_pipe core, which includes an sd_card_emulator, plus some BRAM and FIFOs for use as an emulated MMC device.
3. A main system parellel bus, with automatic baud rate enabled async_syscon system controller, which communicates with registers and BRAM and/or FIFOs in both the host and the data pipe cores. The system controller is a “hardware debugger” unit, which implements a minimal command set via an asynchronous serial communications terminal.
4. An mmc_test_cmd_rx, which can “snoop” on the MMC command bus. The timestamped output of the MMC command snooper is buffered in a small FIFO, and then periodically sent out for archiving elsewhere by means of an asynchronous serial transmitter connected to a separate serial port (This is essentially the transmit half of a UART.) Statistics counters are also provided tracking the number of good messages, the number with CRC errors, and the number with stop bit errors.

5. A set of I/O muxes, which can be controlled via registers accessible over the main bus, or by physical switches on the FPGA board, allowing "crossbar steering" of the MMC bus. By means of these muxes, the internal MMC host can be used with an external MMC card, or the external MMC bus can be used with the internal mmc_data_pipe core, or the internal MMC host and the internal MMC data pipe core can be used with each other.

The sheer scope of this mmc_tester core makes it difficult to explain fully, but with careful understanding of its components, the purpose and usefulness of the different settings and facilities it provides should become clear. It also aids understanding to study the memory map of the items connected to the system bus which is controlled by the hardware debugger.

The block diagram of this core is:



For simplicity, the register connections, and the "flancters" used for safe clock domain crossing are not shown in this block diagram.  There are two main clock domains in this core, the first being the SD/MMC cardbus clock domain, which is used by substantial parts of the MMC host and MMC slave cores, and the second being the FPGA system clock, which is used by the rest of the system, including the async_syscon system controller, the telemetry sender, and all the registers.

A compact summary of the mmc_tester memory map is given here:

| Address | Length | Function |
|---|---|---|
| 0x03000000 | 0x10 | SD/MMC host core registers |
| 0x03000010 | 0x10 | SD/MMC slave core registers |
| 0x03000020 | 0x10 | mmc_tester core registers |
| 0x04000000 | 0x4000* | 16k byte SD/MMC host BRAM (*Can be set by generics) |
| 0x05000000 | 0x4000* | 16k byte SD/MMC slave BRAM (*Can be set by generics) |

*Table 2: mmc_tester memory map "30,000ft overview"*

The summary of the mmc_tester memory map is a good introduction to the way the components are mapped on the main system bus, but to be more helpful, the expanded memory map is given here. Note that further details on the different fields contained in SD/MMC host and slave registers are given in the sd_card_pack and sd_host_pack documents.  The mmc_tester core registers are further explained using diagrams inside this document.

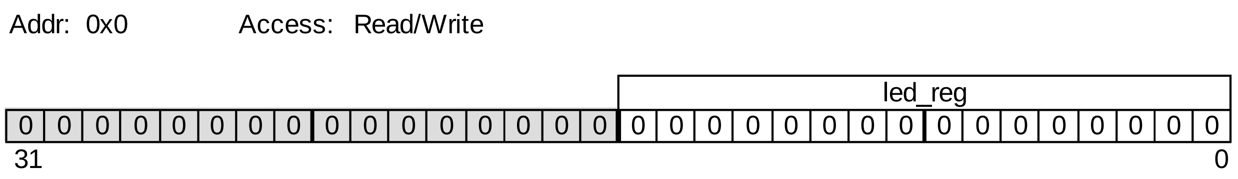
| Address | Location | Function |
|---|---|---|
| 0x03000000 | SD/MMC host | SD/MMC transfer data block size |
| 0x03000001 | SD/MMC host | SD/MMC transfer data block count |
| 0x03000002 | SD/MMC host | SD/MMC command index |
| 0x03000003 | SD/MMC host | SD/MMC command argument |
| 0x03000004 | SD/MMC host | Response register 0 |
| 0x03000005 | SD/MMC host | Response register 1 |
| 0x03000006 | SD/MMC host | Response register 2 |
| 0x03000007 | SD/MMC host | Response register 3 |
| 0x03000008 | SD/MMC host | SD/MMC bus size, reset, bustest and timeout settings |
| 0x03000009 | SD/MMC host | SD/MMC bus operating frequency |
| 0x0300000A | SD/MMC host | SD/MMC command interrupt status register |
| 0x0300000B | SD/MMC host | SD/MMC command interrupt enable register |
| 0x0300000C | SD/MMC host | SD/MMC data interrupt status register |
| 0x0300000D | SD/MMC host | SD/MMC data interrupt enable register |
| 0x0300000E | SD/MMC host | DMA address register |
| 0x0300000F | SD/MMC host | (Reserved) |
| 0x03000010 | SD/MMC slave | Card reported status |
| 0x03000011 | SD/MMC slave | Card RCA and DSR registers |
| 0x03000012 | SD/MMC slave | Card EXT_CSD address |
| 0x03000013 | SD/MMC slave | Card EXT_CSD data |
| 0x03000014 | SD/MMC slave | Card CSD(31:0) |
| 0x03000015 | SD/MMC slave | Card CSD(63:32) |
| 0x03000016 | SD/MMC slave | Card CSD(95:64) |
| 0x03000017 | SD/MMC slave | Card CSD(127:96) |

| | | |
|---|---|---|
| 0x03000018 | SD/MMC slave | (Reserved) |
| 0x03000019 | SD/MMC slave | (Reserved) |
| 0x0300001A | SD/MMC slave | (Reserved) |
| 0x0300001B | SD/MMC slave | (Reserved) |
| 0x0300001C | SD/MMC slave | (Reserved) |
| 0x0300001D | SD/MMC slave | (Reserved) |
| 0x0300001E | SD/MMC slave | (Reserved) |
| 0x0300001F | SD/MMC slave | (Reserved) |
| 0x03000020 | mmc_tester | LED register |
| 0x03000021 | mmc_tester | switch input register (read only) |
| 0x03000022 | mmc_tester | Host/slave enables, telemetry FIFO register access enable |
| 0x03000023 | mmc_tester | mmc_test_cmd_receiver command filter setting |
| 0x03000024 | mmc_tester | received command good count |
| 0x03000025 | mmc_tester | received command CRC bad count |
| 0x03000026 | mmc_tester | received command stop error bad count |
| 0x03000027 | mmc_tester | data block transfer count |
| 0x03000028 | mmc_tester | MMC data bus size setting (write to reset) |
| 0x03000029 | mmc_tester | telemetry FIFO fill level (write to clear FIFO contents) |
| 0x0300002A | mmc_tester | telemetry FIFO data read port (byte-wise shift register) |
| 0x0300002B | mmc_tester | (RESERVED) |
| 0x0300002C | mmc_tester | (RESERVED) |
| 0x0300002D | mmc_tester | (RESERVED) |
| 0x0300002E | mmc_tester | SD/MMC data pipe (internal slave) FIFO clear bits |
| 0x0300002F | mmc_tester | SD/MMC data pipe (internal slave) FIFO read/write port |

*Table 3: mmc_tester memory map full register list*

For each of the registers implemented within the mmc_tester, further explanation and a register diagram are given.  Note that in this core, the parallel system bus is a 32-bit data and 32-bit address bus, with each function being allocated a block of 16 register addresses.  Addresses in the system are 32-bit “double word” addresses, meaning that each address selects a specific 32-bit data word location, and there are no byte-enables or other byte addressing constructs.  the address given for each register is the relative offset from the base address used to generate the 16-register block select signal:

## Register 0x0 : LED Register



Address 0x0300_0020 is R0, the LED outputs register (READ/WRITE).  This register contains a number of bits meant for controlling LEDs on the test board, just for fun.  The sixteen bits in this register are connected to the LEDs on the ARTY board.  The four least significant bits are connected to the green LEDs, and the twelve most significant bits are connected to the RGB LEDs, as shown in Table 4.  Note that the LEDs are in reality driven by a data selector which can select either the LED outputs register or the upper 32-bits of a 40-bit counter which runs at the ARTY system clock rate of 100 MHz.  The selector is controlled by switch SW3, such that if SW3 is on the “OFF” position (slider handle positioned near the edge of the ARTY board) then the LEDs are all driven by the LED outputs register.  On the other hand, if SW3 is “ON” then the LEDs are driven by the counter, which makes for a nice visual display and a direct confirmation that the ARTY board is running the target design.

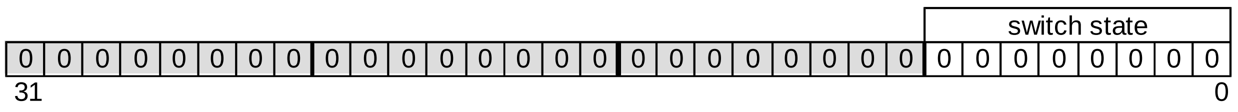
An observer with delicate sensibilities might pose the question: Why are Xilinx ARTY specific LEDs connected to a register in this mmc_tester core, which is meant to be independent of any particular FPGA development board?  The answer is that most FPGA development boards have some set of LEDs available for display purposes, and so the number of bits contained within this register is parameterized so that the user can easily connect it to any given FPGA development board, and drive up to 32 LEDs with it.

| Bit | Connection |
|---|---|
| 0 | LD4 anode, green |
| 1 | LD5 anode, green |
| 2 | LD6 anode, green |
| 3 | LD7 anode, green |
| 4 | LD3, red |
| 5 | LD3, green |
| 6 | LD3, blue |
| 7 | LD2, red |
| 8 | LD2, green |
| 9 | LD2, blue |
| 10 | LD1, red |
| 11 | LD1, green |
| 12 | LD1, blue |
| 13 | LD0, red |
| 14 | LD0, green |
| 15 | LD0, blue |

*Table 4: ARTY LED connections*

## Register 0x1 : Switch State

Addr: 0x1 Access: Read Only



Address 0x0300_0021 is R1, the Switch State register. The Switch State register is read only, and its contents reflect the status of the ARTY switch inputs, in real time. There is no debouncing or filtering applied to these values. The switches are mapped to bits within the register as shown in Table 5.

| Bit | Connection |
|---|---|
| 0 | SW0 |
| 1 | SW1 |
| 2 | SW2 |
| 3 | SW3 |
| 4 | BTN0 |
| 5 | BTN1 |
| 6 | BTN2 |
| 7 | BTN3 |

*Table 5: ARTY switch connections*

An observer with an easily offended sense of design partition might pose the question: Why are Xilinx ARTY specific switches connected to a register in this mmc_tester core, which is meant to be independent of any particular FPGA development board? The answer is that many FPGA development boards have some set of switches available as inputs to the FPGA, and so the number of bits contained within this register is parameterized so that the user can easily connect it to any given FPGA development board, and receive up to 32 switch states with it.

Note that some of the switches are dedicated to special functions within the mmc_tester design. The special functions are summarized in Table 6.

| Switch | Dedicated Function |
|---|---|
| SW1 | MMC host core enable |
| SW2 | MMC slave core (data pipe) enable |
| SW3 | Drive the LEDs from counter. Also, send any traffic log data as asynchronous bytes. |

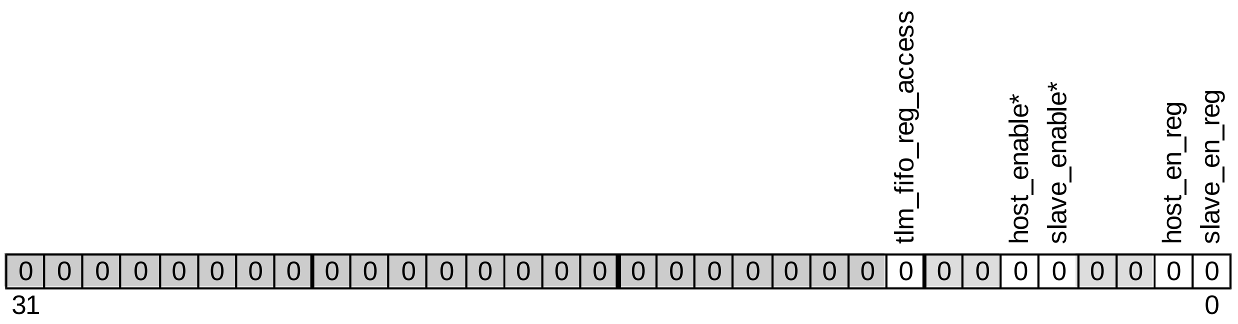
*Table 6: ARTY dedicated switch functions*

The functions of SW1 and SW2 are also available via register bits in register 0x2.

## Register 0x2 : Function Enables

Address 0x0300_0022 is R2, the Function Enables register. Certain functions of the mmc_tester core can be enabled or disabled via setting or clearing bits in the Function Enables register.

Addr: 0x2 Access: Read/Write (* Read Only portions)



**Bit [0]** = Slave Enable Reg. Setting this bit causes the internal MMC_data_pipe slave to be active on the SD/MMC bus.

**Bit [1]** = Host Enable Reg. Setting this bit causes the internal sd_controller_8bit_bram unit to be active on the SD/MMC bus.

**Bits [3..2]** = "00" (Reserved)

**Bit [4]** = Slave Enabled (READ ONLY). This bit is the logical OR of bit[0] and the slave_en_i input, which is connected to ARTY switch SW2. Thus we see that the MMC_data_pipe slave can be made active by the switch, or the register bit.

**Bit [5]** = Host Enabled (READ ONLY). This bit is the logical OR of bit[1] and the host_en_i input, which is connected to ARTY switch SW1. Thus we see that the sd_controller_8bit_bram host unit can be made active by the switch, or the register bit.

**Bits [7..6]** = "00" (Reserved)

**Bit [8]** = Telemetry log register read access enable. When this bit is set, then telemetry log FIFO contents can be read out via register R10 (0xA), and the asynchronous serial transmitter is disabled. When clear, the asynchronous serial transmitter has access to the telemetry log FIFO instead of register R10.

For inquiring minds that want to know, a further explanation of the operation of the enable bits may be helpful:

Register 2, bit[0] : "slave_en_reg"

When high, this bit causes the MMC slave to receive commands, and respond to them. When this bit is enabled, the MMC slave will drive signals onto mmc_cmd_o and mmc_dat_o, including driving the mmc_cmd_oe_o and mmc_dat_oe_o lines when appropriate. When this bit is low, the MMC slave cmd input will be held high, and the MMC slave will not have anything to do.

Register 2, bit[1] : "host_en_reg"

When high, this bit causes the MMC host to send commands, and receive responses to them. When this bit is enabled, the MMC host can drive signals onto mmc_cmd_o and mmc_dat_o, including driving the mmc_cmd_oe_o and mmc_dat_oe_o lines when appropriate. When this bit is low, the MMC host cmd output will be cut off from affecting mmc_cmd_o, and the MMC host will not be able to influence the MMC bus signals.

If both of these bits are low, then the only function active in this core is to monitor the mmc commands which occur on the MMC bus due to the action of any outside MMC hosts, such as card readers, which may send commands to an MMC card on the bus.

With "host_enable" set, the mmc_tester can be used to communicate with a real external SD/MMC device, perhaps for the purpose of reading or writing some sectors, or reading the device registers including the CID, CSD and the 512 byte EXT_CSD.

With "slave_enable" set, on the other hand, an MMC slave can be used with an external SD/MMC card reader. The MMC slave includes a small amount of RAM, plus a pair of FIFO buffers. Writing data to the card at addresses below the upper boundary of the RAM, simply places the data into RAM. Beyond the RAM upper boundary, the data goes into the write FIFO. Reading is handled the same way.

An alternative to using the register bits to enable the slave and host, is to use the input signals "slave_en_i" and "host_en_i", which are connected to ARTY switches. These signals are simply logically ORed with the slave_en_reg and host_en_reg register bits, to produce the read only bits “slave_enable” and “host_enable.”

## Register 0x3 : Test Command Receiver Filter

Address 0x0300_0023 is R3, the Test Command Receiver Filter register. This register is a filter which can serve to pass a desired set of commands through to the command log, rejecting all the rest. It is useful for homing in on a particular command of interest.

Addr: 0x3 Access: Read/Write

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | | | t_rx_cmd_filter | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

**Bits [7:0]** determine what type of command creates a "command event" for capture in the log. Except for the value of 0x80, setting bit [7] means "capture and count everything," including all command indices, in both directions.

Example settings:

0x81..0xFF => Capture and count everything.

0x80 => Capture and count nothing. Nada. Zilch.

0x7F => Capture and count all host-to-card commands

0x3F => Capture and count all card-to-host responses

0x59 => Capture and count CMD25 from host-to-card only

0x28 => Capture and count CMD40 R5 responses only

0x27 => Capture and count CMD39 R4 responses only

0x19 => Capture and count CMD25 R1 responses only

0x00 => Capture and count CMD0 R1 responses only

To explain more clearly how these filter settings work, it is helpful to recall that the first eight bits of the command or response are:

[0][d][cccccc]

Where:

0 = start bit (always low)

d = direction bit (1 for host to card, 0 for card response)

cccccc = command index, ranging from 0 to 56.

A special setting of 63 for the cccccc field, selects long responses and/or 48-bit R3 OCR responses. Therefore, since 136 bit long responses are not differentiated from 48-bit R3 OCR responses, it becomes clear that the design of the filter is not perfect.  For example, there is no way to act on only R3 responses.  However, there is really no need for a perfect filter.  This one is, well, you know, good enough.

## Registers 0x4 – 0x7 : Transfer statistics registers

These four registers contain counts of different categories of transfers detected on the SD/MMC bus.

Addr: 0x4 Access: Read/Write (Writing clears to zero)

| t_rx_cmd_gdcount | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Addr: 0x5 Access: Read/Write (Writing clears to zero)

| t_rx_cmd_crc_bdcount | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Addr: 0x6 Access: Read/Write (Writing clears to zero)

| t_rx_cmd_stp_bdcount | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Addr: 0x7 Access: Read/Write (Writing clears to zero)

| t_rx_dat_count | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Address 0x0300_0024 (R4) is the Error-free Command Events count.

Address 0x0300_0025 (R5) is the Command Events with CRC error count.

Address 0x0300_0026 (R6) is the Command Events with stop bit error count.

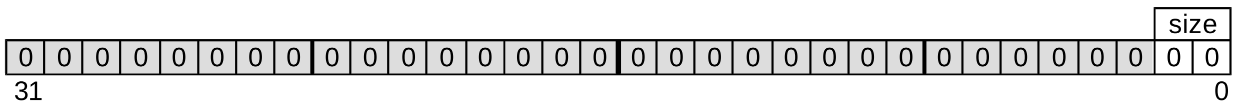
Address 0x0300_0027 (R7) is the Number of data transfers completed count.

No clear means are provided for the mmc_tester to determine what direction the data is flowing over the SD/MMC data lines. Therefore, the R7 register simply contains the total number of transfers started in either direction. Note that this 32-bit counter is distinct from the eight bit "tlm_d_count" data transfer counter in the telemetry, which is a relative count, cleared at every command transfer event.

## Register 0x8 : SD/MMC data bus size register

This register allows the user of the mmc_tester to read the current SD/MMC data bus size. This register setting can only be altered to larger bus sizes by the action of the SD/MMC host, sending a SWITCH command. However, through the system bus interface, it can be cleared back to its default setting of “00.”

Addr: 0x8 Access: Read/Write (Writing clears to zero)



Address 0x0300_0028 is R8, the reg_dbus_size register.

**Bits [1:0]** reflect the current SD/MMC data bus size setting, as follows:

0 => 1 bit data bus transfers

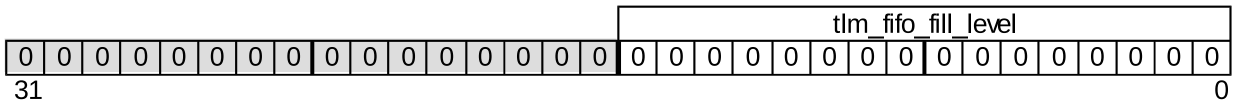
1 => 4 bit data bus transfers

2 => 8 bit data bus transfers

In order for the mmc_tester to determine when a data transfer is completed, so that it can count valid start bits, it is required to know the data bus size. Logic is present to decode when a host-to-card command of index 6 (CMD6), the SWITCH command, is seen with an argument 0x03B70s00, where s contains the new data bus size. This 2-bit setting is then stored in the reg_dbus_size register, and decoded to set the "dstart_wait" parameter used for ignoring data bus activity during an active data transfer. The reg_dbus_size starts out at "00b", and is expected to change to "01b" for 4-bit SD/MMC activity, and to "10b" for 8-bit MMC activity. SD cards are restricted to 4-bit data bus width maximum.

## Register 0x9 : Traffic log FIFO fill level

Addr: 0x9 Access: Read/Write (Writing clears to zero)



This register allows the user of the mmc_tester to read the number of entries currently present in the traffic log, or “telemetry stream” FIFO. The size of this FIFO is currently set by a constant within the mmc_tester core at 16384 bytes. The FIFO receives and stores time stamped entries, each entry representing a command or data transaction detected on the SD/MMC bus. The format of the entries is explained further in the description of register 0xA.

## Register 0xA : Traffic log FIFO data

Addr: 0xA Access: Read Only

| tlm_r10_dat | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

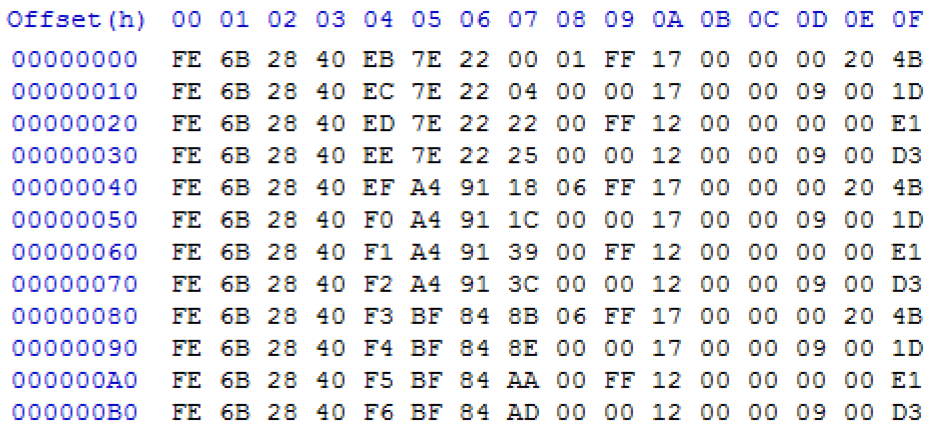
Address 0x0300_002A is R10, the SD/MMC traffic log FIFO read port. Telemetry bytes can be read out of the traffic log FIFO whenever enabled in R2 bit 8. For convenience in reading the data, the bytes are packed into 32-bit words. If R2 bit 8 is clear, then reading this register returns 0x55555555.

Each traffic log (A.K.A. telemetry stream) entry is composed of 16 bytes, shown in Table 7.

| Slice Bytes | Payload | Name | Contents |
|---|---|---|---|
| [0..3] | 32 bits | Sync Pattern | 0xFE6B2840 |
| [4] | 8 bits | tlm_fid | Frame ID, a number which increments with each frame. |
| [5..7] | 24 bits | tlm_tstamp | Timestamp, in µsec. Has a 16.777216 s rollover period. |
| [8] | 8 bits | tlm_d_count | Data transfer count, cleared at every host command. |
| [9] | 8 bits | tlm_code_byte | 255 for host-to-card commands, 0 for card-to-host. |
| [10..15] | 48 bits | t_rx_cmd_raw | Raw contents of the detected command. |

*Table 7: mmc_tester traffic log record format*

The captured traffic log can be output via an asynchronous transmitter. If the tlm_fifo_reg_access bit is clear in register 0x2, and ARTY SW3 is turned on, the regular asynchronous serial response stream is immediately preempted by a stream of telemetry data spewing forth for capture on a host computer system. The binary data bytes are sent out using the same Baud rate as the established hardware debugger serial session, and there is no flow control, so the receiving software must be ready to buffer and store the data immediately. An example of some SD/MMC bus traffic log data captured into a file, as displayed in a binary editor, is shown in Figure 1.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FE 6B 28 40 EB 7E 22 00 01 FF 17 00 00 00 20 4B
00000010  FE 6B 28 40 EC 7E 22 04 00 00 17 00 00 09 00 1D
00000020  FE 6B 28 40 ED 7E 22 22 00 FF 12 00 00 00 00 E1
00000030  FE 6B 28 40 EE 7E 22 25 00 00 12 00 00 09 00 D3
00000040  FE 6B 28 40 EF A4 91 18 06 FF 17 00 00 00 20 4B
00000050  FE 6B 28 40 F0 A4 91 1C 00 00 17 00 00 09 00 1D
00000060  FE 6B 28 40 F1 A4 91 39 00 FF 12 00 00 00 00 E1
00000070  FE 6B 28 40 F2 A4 91 3C 00 00 12 00 00 09 00 D3
00000080  FE 6B 28 40 F3 BF 84 8B 06 FF 17 00 00 00 20 4B
00000090  FE 6B 28 40 F4 BF 84 8E 00 00 17 00 00 09 00 1D
000000A0  FE 6B 28 40 F5 BF 84 AA 00 FF 12 00 00 00 00 E1
000000B0  FE 6B 28 40 F6 BF 84 AD 00 00 12 00 00 09 00 D3
```

*Figure 1: SD/MMC traffic log file example*

## Register 0xD : mmc_data_pipe write FIFO fill level

Addr: 0xD Access: Read/Write (Writing clears the entire FIFO)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | s_fif_dat_wr_level | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Address 0x0300_002D is R13, the mmc_data_pipe write data FIFO fill level. Reading this register returns the number of bytes in the MMC slave write data FIFO. The FIFO holds data that are meant to flow from the MMC slave to the host. Writing to this register clears the MMC slave write data FIFO.

## Register 0xE : mmc_data_pipe read FIFO fill level

Addr: 0xE Access: Read/Write (Writing clears the entire FIFO)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | s_fif_dat_rd_level | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Address 0x0300_002E is R14, the mmc_data_pipe read data FIFO fill level. Reading this register returns the number of bytes in the MMC slave read data FIFO. The FIFO holds data that have been sent from the host to the MMC slave. Writing to this register clears the MMC slave read data FIFO.

## Register 0xF : mmc_data_pipe FIFO data

Addr: 0xF  Access: Read/Write (Reads from RX fifo, Writes to TX fifo)

| | | | | | | | | | | | | | | | | | | | | | | | | Slave FIFO port | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

Address 0x0300_002F is R15, the MMC_data_pipe FIFO data port. Writing to this address loads another byte into the MMC slave write data FIFO, thereby enqueueing it to be read by the host from the MMC slave. However, if the write data FIFO is full, then nothing happens, and the data bits are thrown into the "bit bucket." Reading from this address reads a byte, removes it from the MMC slave read data FIFO, which data was previously delivered from the host to the MMC slave. However, if the read data FIFO is empty, then no valid data is actually delivered, as can well be understood.