Guidelines For Optimal Use Of eMSI on ADSP-SC598 SHARC+ Processor Family

Contributed by Sammit Joshi and Nabeel Shah

Rev 01 – October 4, 2023

Introduction

The ADSP-SC598 processor incorporates highly configurable eMMC/SD host controller called Enhanced Mobile Storage Interface (eMSI), hereby referred to as eMSI or eMSI controller. Both eMMC and SD cards can be interfaced with eMSI. The embedded Multimedia card (eMMC) device is a *NAND flash* memory-based storage device with an integrated controller. The eMMC device is available in various densities and is cost-effective compared to NOR flash devices with comparable read and write performance. eMMC is widely used in the automotive industry for audio infotainment and navigation systems. SD cards or SD is a proprietary non-volatile flash memory card format developed by the SD Association (SDA). SD card is used in various portable devices, such as car navigation systems, and cellular phones.

This EE Note provides recommended guidelines for using an eMSI controller for seamless operation and optimized performance. The EE note discusses silicon anomalies and how to configure the controller to work around them to get the reliable and optimal performance. It also talks about different operating modes of the eMSI and how to use them efficiently to get best performance and minimize latency for the accesses to eMMC and SD card devices.

For more details about the eMSI, refer to the *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference* [1].

Configuring eMSI to Work Around Silicon Anomalies

There are some silicon anomalies associated with eMSI module of ADSP-SC598 processor. It is important that user is aware of these anomalies and measures in application software are taken to ensure reliability in data transactions over eMSI. These silicon anomalies are in the *ADSP-SC595/SC596/SC598 Anomaly List* [2].

One such anomaly which deserves some attention, *Anomaly#20000119 – END bit error may occur during eMSI data transfers due to clock gating*. This anomaly occurs when the eMSI bus clock (clock to eMMC device/SD card) is gated due to FIFO full condition or deliberate programming of the eMSI controller to stop the eMSI bus clock during multiblock read data transfer.

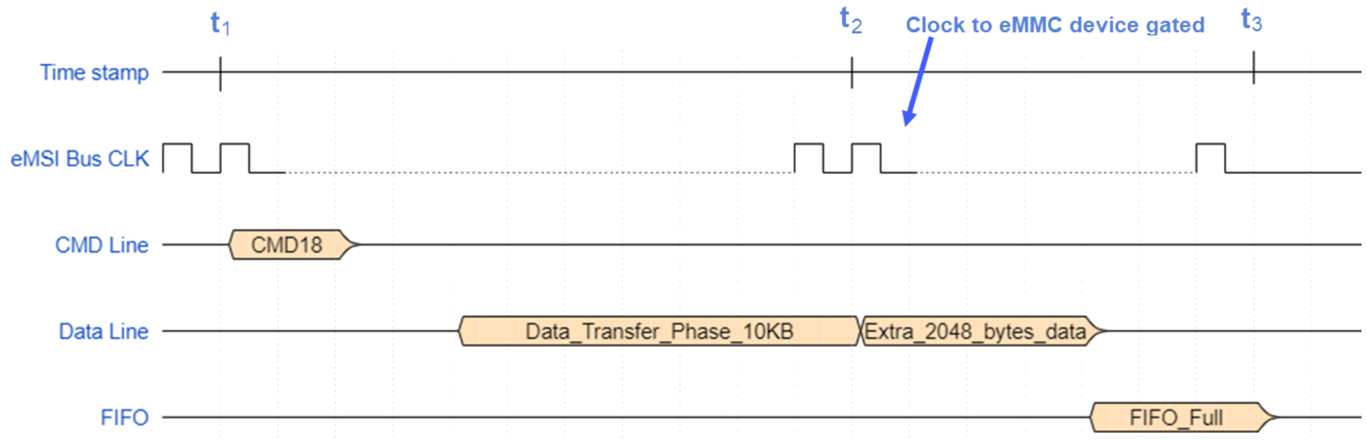
The eMSI FIFO has a depth of 2048 bytes. During multi-block read transfers when a user does not provide enough descriptors or deliberately tries to stop read transfers by stopping the eMSI bus clock then the eMSI controller raises END bit error due to incorrect sampling of the data block END bit by the eMSI controller.

As shown in Figure 1, we have a free running eMSI clock, at time instance **t₁**, assuming that all the necessary configuration from the eMMC device and controller side such as descriptor and register configuration is completed for read transfers.

Assume that the user has configured the descriptor to accommodate 10 kB of data, at time instance **t_1** user issues multiblock read command (CMD18) for reading data from the eMMC device, and CMD23 (SET_BLK_COUNT) was not issued before CMD18 (This results in open-ended transfer). At time instance **t_2** user gets notified about the transfer is completed, after 10 kB of data is received. The user should issue a CMD12 (STOP_TRANSFER) command to stop open-ended transfer as there is no extra descriptor (buffer space in memory) configured to accommodate incoming data from the eMMC device. When the user does not stop the transfer using CMD12 then the eMMC device keeps sending the data and FIFO gets full after the FIFO limit is reached (2048 Bytes). FIFO full condition occurred at time instance **t_3**. As there is no further space to accommodate the incoming data, the eMSI controller gates the eMSI bus clock (at time instance **t_3**) to the eMMC device so that the eMMC device stop driving the data back to the eMSI controller.

Due to clock gating END bit of the last data block gets incorrectly sampled and resulting in an END bit error and stopping further data transfers. To restart the activity controller should be restarted fully.

Figure 1: Clock Gating Scenario Where FIFO is Full Due to Insufficient Descriptor

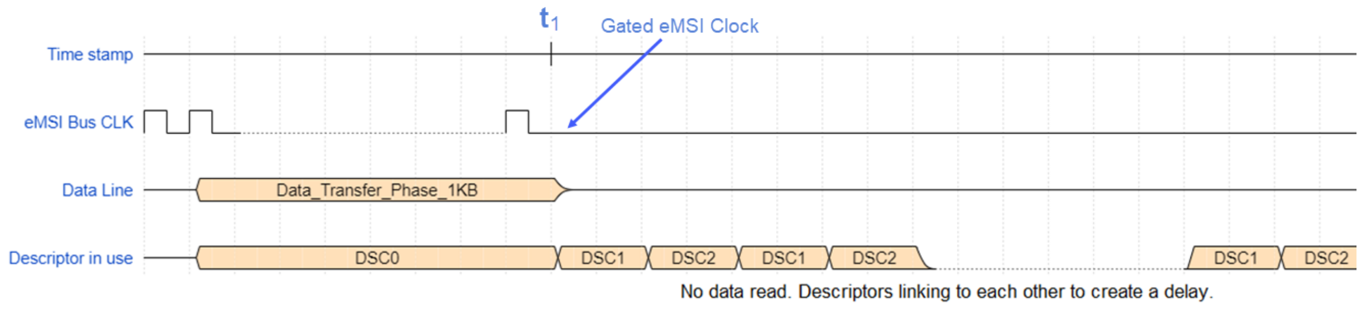


Consider the scenario where user want to read all data with single multi block read but control the data flow from the eMMC device using a eMSI bus clock gating. As shown in Figure 2, when the CMD 18 (single multi block read) is sent to the eMMC device and the eMSI controller is receiving data from eMMC device. There were three descriptors declared namely DSC0, DSC1, and DSC2. DSC0 acted as a transfer descriptor (configure to accommodate 1kB of data), whereas DSC1 and DSC2 are link descriptors. At the time instance **t_1**, eMSI controller has received 1 kB of data and the user wants to add some delay before receiving the next set of data.

To create this delay user deliberately loop back link descriptors, as the transfer descriptor is not available eMSI controller will stop the read data transfer by gating the clock.

Due to clock gating END bit of the last data block gets incorrectly sampled and resulting in an END bit error and stopping further data transfers. To restart the activity controller should be restarted fully.

Figure 2: Deliberated Clock Gating by Introducing a Delay in Data Receiving



Note: Above examples are explained with the eMMC device. Same explanation applies to SD cards.

For more details about the transfer and link descriptors, refer to the eMSI chapter in the *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference* [1].

To avoid the issue due to the clock gating below are some of the recommendations that users can adhere while performing the multi-block read data transfers, These measures will ensure that there is no clock gating in between an active eMSI data transfer and thus will avoid the end bit error issue as described above.

Usage of Single Block Read Command

As mentioned above, the End bit error occurs due to no space in FIFO for received data from SD/eMMC device, due to which the eMSI controller gates the eMSI bus clock to stop receiving data from the SD/eMMC device this leads to the END bit error being raised.

This issue can be avoided altogether when the received data size is less than the depth of FIFO (2048 bytes) so that the FIFO is never full during the read operation. Using the single block read command (CMD17) user can read a block of a size selected by the SET_BLOCKLEN (CMD16) from the SD/eMMC device. Most of the SD/eMMC devices support a maximum block size of 512 bytes (Please refer to `READ_BL_LEN[83:80]` of CSD register from *JEDEC eMMC 5.1 Specification - JESD84-B51* [3] for eMMC device or *SD Specifications - Part 1 Physical Layer Specification* [4] for SD cards) which is less than FIFO depth 2048 bytes hence it will eliminate the FIFO full condition. Users can divide the read transfer into multiple single-block transfers to eliminate this issue. This workaround is applicable in both SDR (for SD/eMMC) and DDR (only for eMMC) speed modes of operation.

Usage of eMSI Tuning Logic

The usage of a single block read resolves the issue, but it takes a great hit on throughput performance as each read command adds initial latency which reduces the overall throughput. To avoid this loss of throughput, using a multi-block read is still preferred as it incurs the initial latency only once thus improving the throughput. However, with multi-block read, to control the flow of data from the card, gating the clock is required. Thus, to avoid the end-bit error in such a case, there is an option within the eMSI controller to tune the clock delay path (eMSI tuning logic) inside the chip to mitigate the issue. When clock gating happens the eMSI controller stops the eMSI bus clock after the ongoing block transfer is completed. This last data block END bit gets sampled incorrectly due to which END bit error gets raised.

The sampling of the END bit can be corrected by enabling eMSI tuning logic. This logic uses tuning circuitry to adjust the clock internally to sample the END bit correctly.

Tuning logics gates the clock signal after negative edge of the clock instead of after positive edge of the clock as shown in figure. eMSI tuning logic can be enabled as shown in the below code:

```
PADS0_PCFG0 |= BITM_PADS_PCFG0_EMSI_TUNING_EN;
PADS0_PCFG1 |= ENUM_PADS_PCFG1_INV4;
```

However, there is limitation that the tuning logic can be used only in SDR speed mode and does not fix the end-bit error issue in DDR speed mode.

Note: eMSI tuning logic is supported for eMMC devices, not for SD cards.

Increasing eMSI Quality of Service (QoS)

Above case assumes that user is intentionally gating the eMSI clock to control the read of the data in multi-block read. It assumes that there is always enough system bandwidth available at SoC system AXI bus for eMSI data to be read from the eMSI internal FIFO and transferred to system memory.

However, there are cases where even though user does not intend to gate the eMSI clock, but the clock gating may still happen when the eMSI FIFO gets full with incoming data and the system bus is unable to transfer the data to system memory in time due to some other higher priority transactions going on AXI bus like some other high bandwidth memory to memory transfers that can hog up the AXI bus depriving eMSI of required bandwidth. In such case due to FIFO getting full, eMSI clock may get gating for some time until FIFO is freed up. This may again result in end bit error due to clock gating.

As example case is when multiple DMAs are operating (Peripheral DMA + Memory DMA) at a system level, eMSI might not get enough bandwidth during read transfers for transferring data between eMSI FIFO to core internal/external memory. This may cause FIFO full condition which can lead to END bit error. The error is more prominent when destination buffers for DMA and eMSI are in L3 memory.

Such a situation can be avoided when eMSI priority is increased to ensure that even in presence of other system transaction, eMSI always gets the required bandwidth on priority. This can be achieved when eMSI QoS in SCB is *increased* from seven to twelve then eMSI gets appropriate bandwidth at the system level. This will avoid eMSI FIFO getting full and consequentially avoiding END bit errors.

Here is the programming step to configure QoS for eMSI:

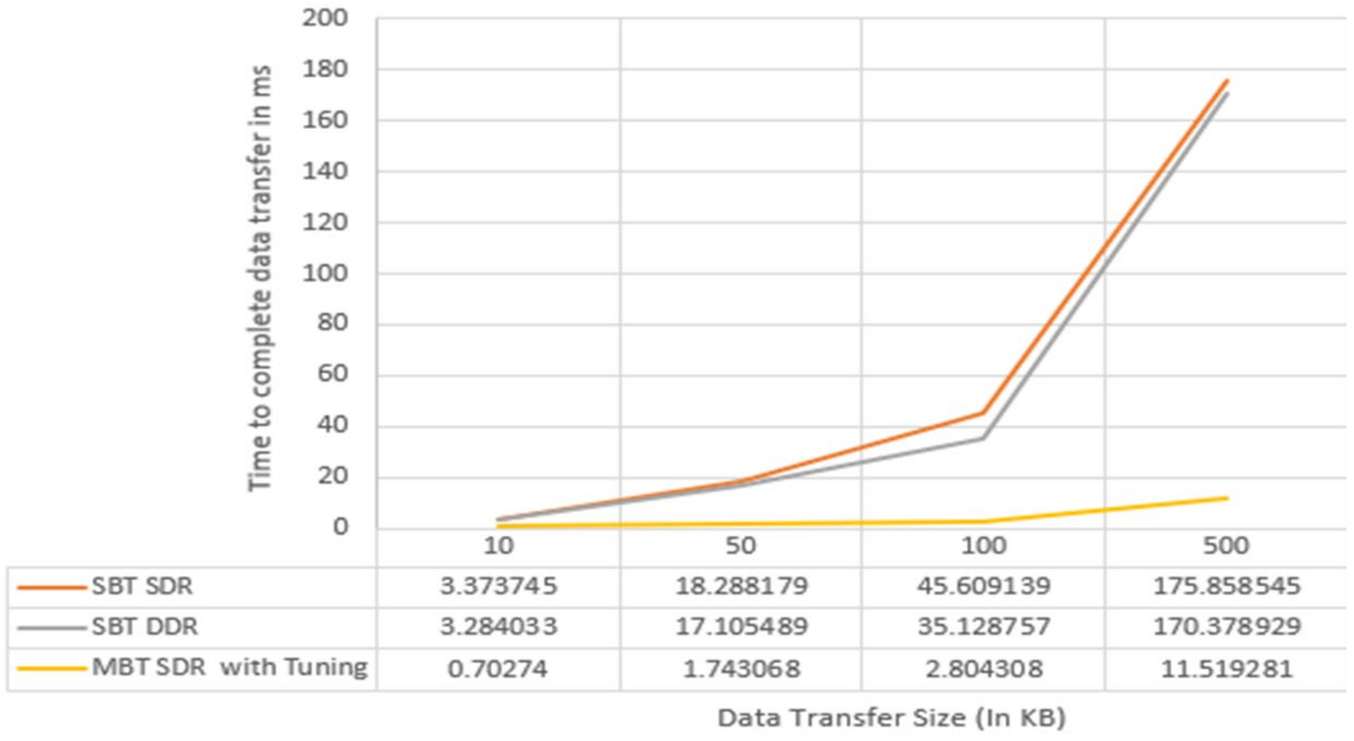
```
SCB0_SDIO0_IB_WRITE_QOS = 0xC;
```

The above workaround is applicable for both eMMC devices and SD cards. This workaround won't help when the eMSI bus clock is deliberately gated (for example by providing insufficient transfer descriptors).

Performance Comparison Between Different Workarounds

This section compares performance of single block data transfer vs multiblock data transfer with tuning working enabled. Please refer to Figure 3.

Figure 3: Performance Comparison For Different Workarounds



	10	50	100	500
SBT SDR	3.373745	18.288179	45.609139	175.858545
SBT DDR	3.284033	17.105489	35.128757	170.378929
MBT SDR with Tuning	0.70274	1.743068	2.804308	11.519281

In the experiment, various payload size read operations were performed, and the user waits in the application till all transfers are completed. Single block read transfer in SDR speed mode (SBT SDR) takes around 3.37 ms, Single block read transfer in DDR speed mode (SBT DDR) takes around 3.284033 ms for 10 KB of transfers whereas multiblock transfer in SDR mode (with eMSI tuning logic enabled) takes around 0.702740 ms, which is 4 times improvement compared to SBT SDR and SBT DDR. As transfer size increases this improvement is more significant as we can see for a 500 KB data transfer, the size improvement is 15 times.

So using multiblock read transfer in SDR speed with eMSI tuning logic enabled gives the best performance and eliminates raising of END bit error in case of eMMC devices (for SD card eMSI tuning logic is not applicable).

Note: All data was captured with the IS22ES08G-JCLA1 eMMC device present on EV-SC598-SOM-EZKIT at CCLK (Core clock) = 1 GHz and eMSI bus clock = 50 MHz (8 bit)

General Guidelines For eMSI Transfers

Configuration of eMSI controller should be done properly before performing multiblock read transfers such as configuring DMA and register with proper values. This section contain guidelines that can be followed while configuring eMSI controller for seamless eMSI operation during multiblock read operation. When an engineer avoids these guidelines then an END bit error scenario can happen as explained above.

There are two types of data transfers available for eMMC devices/SD cards:

- Predefined Transfers (Transfers with CMD23): The SD/eMMC device will transfer the requested number of data blocks, terminate the transaction, and return to the transfer state. In the case of SD cards, predefined transfers (CMD23 support) are only supported above version 3.00 cards. Please refer to the SCR register section of the *SD Specifications - Part 1 Physical Layer Specification* [4] for more information on SD card version.
- Open- Ended Transfers (Transfers without CMD23): The number of blocks for the read multiple block operation is not defined. The device will continuously send data blocks until a stop transmission command (CMD12) is received.

eMMC Device-specific Guidelines

In both cases, when the selected DMA is ADMA, then the user should provide enough transfer descriptors and when the selected DMA is SDMA, then the System Address Register (`EMSI_SDMA_ADDR/EMSI_ADMA_ADDR_LO`) shall be updated as soon as DMA interrupt status bit (`EMSI_ISTAT_DMA_INTERRUPT`) is set.

For Open ended transfer, where `EMSI_BLKCNT.VALUE` is greater than 0 and `EMSI_TRNSFRMODE. BLOCK_COUNT_EN` is enabled, then the user should terminate the transfers using STOP_TRANSMISSION (CMD12) as soon as the transfer complete status bit (`EMSI_ISTAT_XFER_COMPLETE`) is set for both DMA modes. And when `EMSI_BLKCNT.VALUE` is = 0 and `EMSI_TRNSFRMODE`. BLOCK_COUNT_EN is disabled, then the user should ensure that the stop transmission command is sent before all receive buffers (memory space) is full, as the transfer complete status will not get set in this case or use SDR mode operations with tuning enabled.

Note: When the above guidelines are not followed, then END bit error occurs while performing multiblock read transfers.

SD Card-specific Guidelines

SD cards have various versions for example, version 1.01, 1.10, 2.00, 3.00, 4.XX, etc. Please follow the following version-specific guidelines to avoid END bit error.

Version 1.01/1.10/2.00 SD Cards:

Version 1.01/1.10/2.00 SD cards do not support CMD 23 (SET_BLOCK_COUNT) command. Hence only open-ended transfers are possible with these versions of SD cards.

For these versions of SD cards while performing open-ended transfers user should ensure the following conditions are met:

- When `EMSI_BLKCNT.VALUE` is >0 and `EMSI_TRNSFRMODE. BLOCK_COUNT_EN` is enabled then the user should terminate the transfers using STOP_TRANSMISSION (CMD12) as soon as the transfer complete status bit (`EMSI_ISTAT_XFER_COMPLETE`) is set for both SDMA and ADMA operations. When the guideline is not followed, then END bit error occurs while performing multiblock read transfers.
- When `EMSI_BLKCNT.VALUE` is = 0 and `EMSI_TRNSFRMODE. BLOCK_COUNT_EN` is disabled then the user should ensure that the stop transmission command is sent before all receive buffers (memory space) are full as the transfer complete status will not get set in this case.

- The second condition may result in an END bit error as the user does not get notified about the transfer status, hence for safe operations, in these SD card versions usage of a single block read a transfer command is recommended.

Version 3.00/4.XX and Above SD Cards

Version 3.00/4.XX and above SD card support the CMD 23 (SET_BLOCK_COUNT) command. Hence, open-ended transfers and predefined transfers are possible with these versions of SD cards.

For these versions of cards, the guidelines remain the same as eMMC device-specific guidelines except for eMSI tuning logic support, that is, tuning logic workaround is not supported for SD cards.

As mentioned for less than version 3.00 cards, open-ended transfer conditions may result in an END bit error, hence it is advised to use pre-defined transfers in Version 3.00 and above cards.

Optimizing Performance with Command Queuing-based Transfer

The *JEDEC eMMC 5.1 Specification - JESD84-B51* [3] allows user to do data transfers using two methods, (**1**) Isolated Single or multiblock transfers and (**2**) command queuing-based data transfer. Using the first method a user must write multiple registers and configure descriptors each time before data transfer, which adds launch latency I for the next transfer and thus degrades the overall data transfer performance.

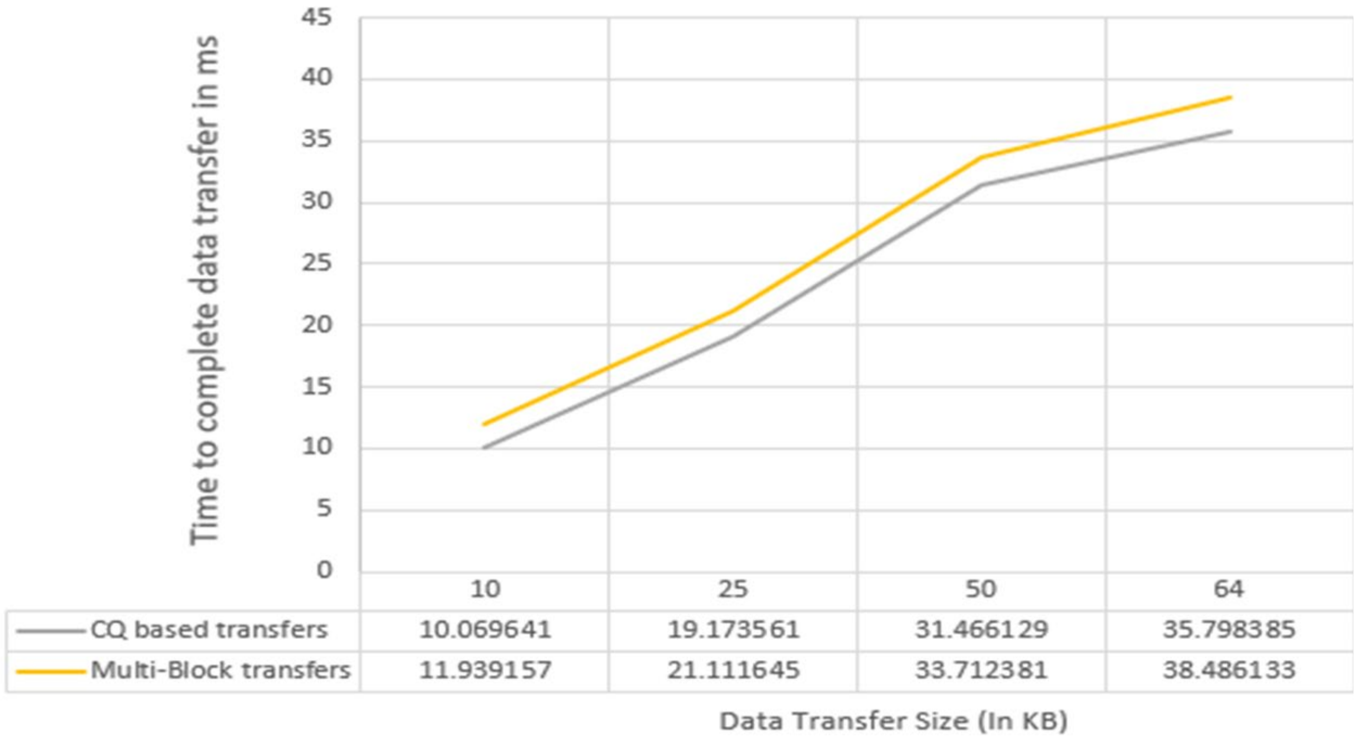
For improving performance eMMC supports command queuing-based data transfers. Users can submit up to 32 data transfer tasks (Hereafter referred to as tasks) to an eMMC device at a time. Command Queuing includes new commands for issuing tasks to the device, ordering the execution of previously issued tasks, and for additional task management functions.

eMSI consist of the Command Queueing Engine (CQE) denotes the hardware unit executing the Command Queueing (CQ) activities. The CQE manages the interface between the host software and the eMMC device, and the data transfers. Each task is configured using two descriptors, task descriptor and transfer descriptor and pair of these descriptor corresponding to each task are stored in Host memory. CQE receives tasks from the software through a Task Descriptor List (TDL) in the host memory and the doorbell register. The application gets notified via interrupt once the specified number of tasks (interrupt coalescing) or all tasks are completed. Also, as an advantage user can queue various tasks to various sector addresses of the eMMC device (for example, task one can write to the first sector address where task three can read from the 100th sector address) whereas, in ADMA-based multiblock transfers for each data transfer, a set of registers and descriptors should be written each time. Due to the task queuing approach throughput improvement is observed.

For more details about the CQE, refer to the *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference* [1].

Consider an experiment where the user must perform six tasks (three write tasks and three read tasks) and the user waits in the application till all the data transfer is finished. Where each task (read/write) does 10 KB or 25 KB or 50 KB or 64 KB of transfer. In the case of command queuing transfer, all tasks are submitted to CQE using TDL and ringing doorbell register, whereas for the multiblock transfer for each task, multiblock read or write command (CMD18/CMD25) is issued. Figure 4 summarizes the performance comparison for Command queuing versus multiblock transfers experiment. The comparison shows that command queuing transfers perform better compared to multiblock transfers.

Figure 4: Performance Comparison of the Same Number of Tasks with Different Payload Sizes

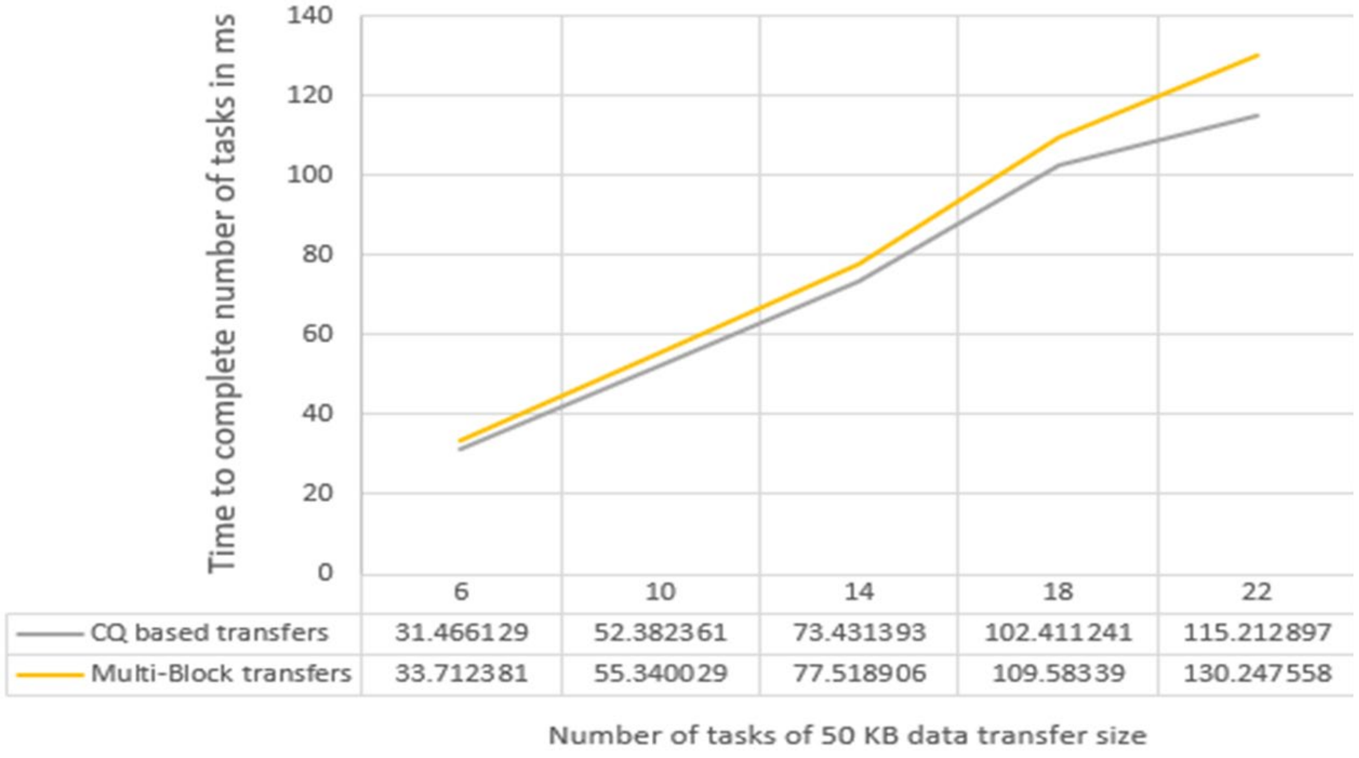


	10	25	50	64
CQ based transfers	10.069641	19.173561	31.466129	35.798385
Multi-Block transfers	11.939157	21.111645	33.712381	38.486133

Consider an experiment like the above experiment, where the user must perform a certain number of tasks (an equal number of read and write tasks) and the user waits in the application till all the data transfer is finished. In this experiment the total number of tasks is varied keeping the payload size for each task at 50 KB. Figure 5 summarizes the performance comparison for this experiment. The comparison shows that command queuing transfers perform better compared to multi-block transfers when the number of tasks increases.

Note: All data was captured with the MX52LM08A11XVI eMMC device @ CCLK = 1 GHz and eMSI bus clock = 25 MHz in 8-Bit DDR speed mode.

Figure 5: Performance Comparison of Different Numbers of Tasks with the Same Payload Size



	6	10	14	18	22
CQ based transfers	31.466129	52.382361	73.431393	102.411241	115.212897
Multi-Block transfers	33.712381	55.340029	77.518906	109.58339	130.247558

Chained Transfer Performance

eMSI supports multiple DMA types such as SDMA, ADMA2, and ADMA3 for the data transfer operations. Where SDMA supports non-descriptor-based transfer, ADMA2, and ADMA3 support descriptor-based transfers. During ADMA2-based data transfer, a single operation can take place at a time [e.g., Single block read (CMD17), single block write (CMD24), Multiblock read (CMD18), or Multiblock write (CMD 25)], before we perform the next read/write transfer, registers and descriptors should be reconfigured.

Similarly for SDMA, some set of registers should be programmed every time before the transfer starts. These latencies for programming registers and descriptors (in the case of ADMA2) add up for long transfers and may degrade the throughput performance.

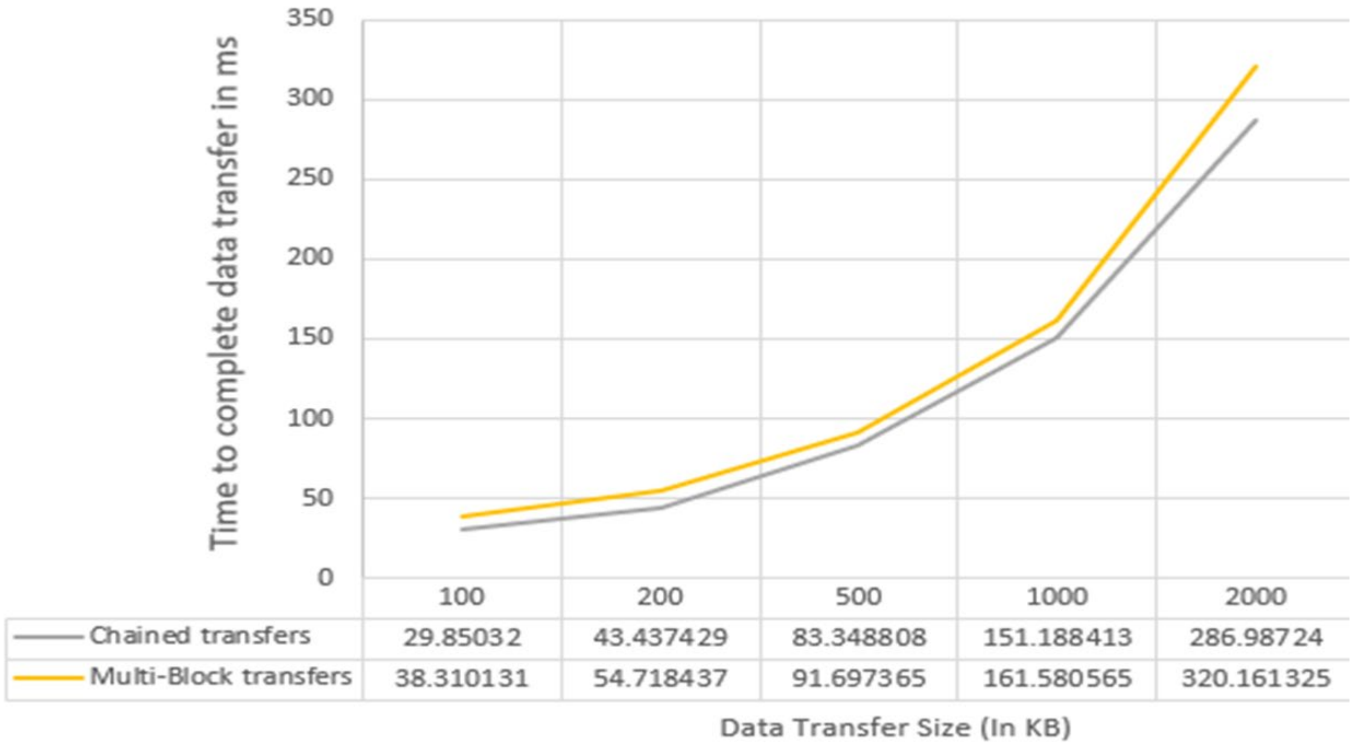
To overcome this issue, eMSI consists of ADMA3. ADMA3 enables the host to program multiple ADMA2 operations. ADMA3 uses Command Descriptor to issue data transfer/non-transfer-based commands (for example, CMD17 and CMD7). A multi-block data transfer between system memory and SD/eMMC device is programmed by using a Command Descriptor and ADMA2 Descriptor pair. ADMA3 performs multiple multi-block data transfers by using an Integrated Descriptor. (Integrated descriptor contains a pointer to the Command Descriptor and ADMA2 Descriptor pairs.)

Using ADMA3, a user can create **n** number of descriptor pairs (Command descriptor and ADMA2 descriptors) and submit multiple data transfer operations for execution. Each transfer gets a completed chained function (one after the other). This improves throughput performance by avoiding writing registers and creating descriptors multiple times.

For more details about the ADMA3, refer to the *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference* [1].

Consider an experiment, where the user must perform six data transfer operations (three write data transfer operations and three read data transfer operations) and the user waits in the application till all the data transfer is finished. Where each write or read operation is of 100 KB or 200 KB or 500 KB or 1 MB or 2 MB. In the case of chained transfer, all six transfers were programmed using command descriptor and transfer descriptor pairs, and the base address of integrated descriptors is given to `EMSI_ADMA_DESADDR_LO` register. This starts all transfer execution in a chained fashion (one after another). For Multi block transfer for each transfer operation Multiblock read or write command (CMD18/CMD25) is issued. Figure 6 summarizes the performance comparison for this experiment. The comparison shows that chained transfers perform better compared to multi-block transfers, for higher payload sizes performance improvement is notable.

Figure 6: Performance Comparison of the Same Number of Transfers with Different Payload Sizes

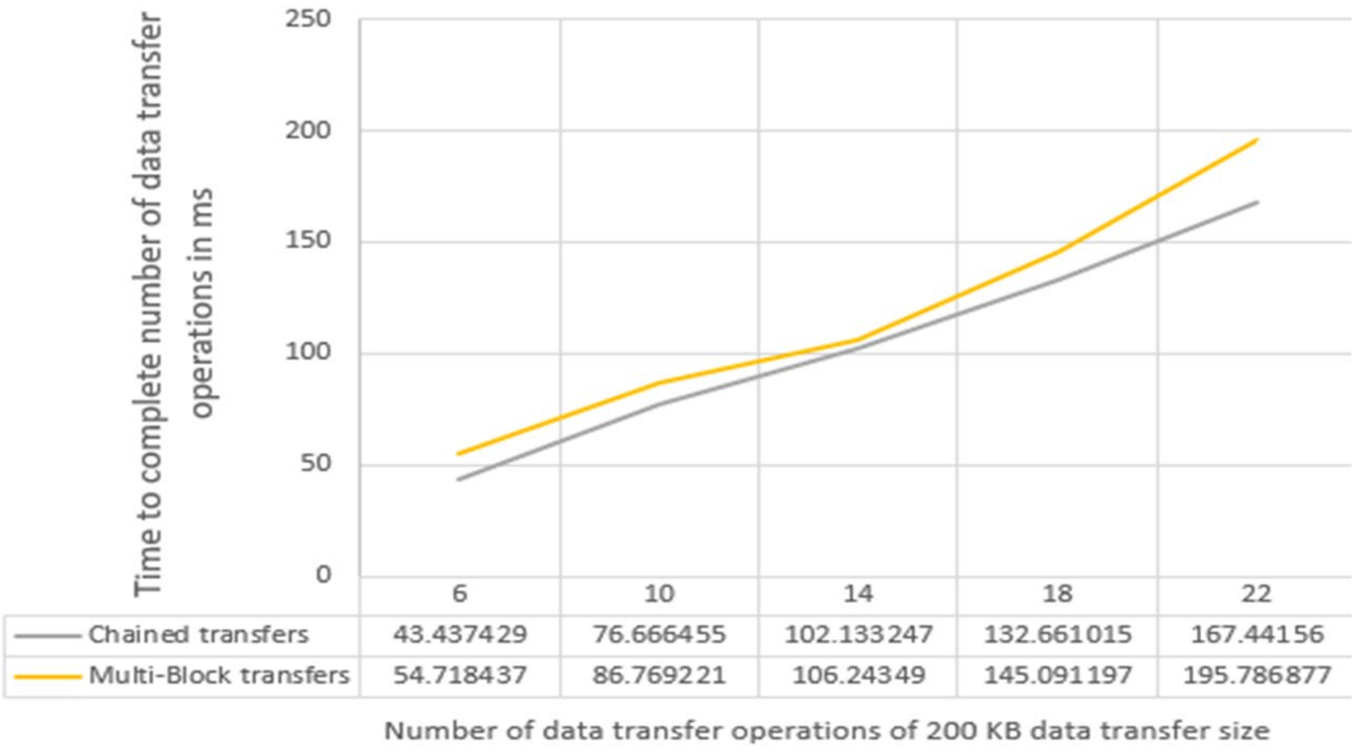


	100	200	500	1000	2000
Chained transfers	29.85032	43.437429	83.348808	151.188413	286.98724
Multi-Block transfers	38.310131	54.718437	91.697365	161.580565	320.161325

Consider an experiment like the above experiment, where the user must perform a certain number of data transfer operations (an equal number of read and write transfer operations) and the user waits in the application till all the data transfer is finished. In this experiment total number of transfers operations are varied keeping the payload size for each task at 200 KB. Figure 7 summarizes the performance comparison for a particular experiment. The comparison shows that chained transfer performs better compared to multi-block transfers as the number of data transfer operations increases.

Note: All data was captured with the IS22ES08G-JCLA1 eMMC device present on a EV-SC598-SOM-EZKIT, @ CCLK = 1 GHz and eMSI bus clock = 50 MHz at 8-Bit DDR speed mode.

Figure 7: Performance Comparison of Different Numbers of Transfers with the Same Payload Size



	6	10	14	18	22
Chained transfers	43.437429	76.666455	102.133247	132.661015	167.44156
Multi-Block transfers	54.718437	86.769221	106.24349	145.091197	195.786877

References

[1] *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference*, Rev 0.2, August 2022, Analog Devices Inc.
https://www.analog.com/media/en/dsp-documentation/processor-manuals/adsp-sc595-sc596-sc598-hrm.pdf

[2] *ADSP-SC595/SC596/SC598 Anomaly List*, Rev. D, November 11, 2022, Analog Devices Inc.
https://www.analog.com/media/en/dsp-documentation/integrated-circuit-anomalies/adsp-sc595-sc596-sc598-anomaly.pdf

[3] *JEDEC eMMC 5.1 Specification - JESD84-B51*, January 2019, Joint Electron Device Engineering Council.
https://www.jedec.org/document_search?search_api_views_fulltext=jesd84-b51

[4] *SD Specifications - Part 1 Physical Layer Specification*, Version 9.00, August 2012, SD Association.
https://www.sdcard.org/downloads/pls/

Document History

Revision	**Description**
Rev 01 – October 4, 2023 *by Sammit Joshi and Nabeel Shah*	Initial Release