A New Data Transfer Scheme for eMMC Connected Subsystems

Shulan Deng, Ph.D
System Software Engineering
Spansion Inc.
Sunnyvale, CA, USA
Shulan.deng@spansion.com

Abstract—One of the issues in data transfer between host CPU and its eMMC connected subsystem is to determine when to send data from host to subsystem and when to receive data from subsystem to host with least CPU interference. The conventional approach to achieving data transfer synchronization is by polling, which impacts CPU bandwidth and potentially affects the subsystem's performance. This paper proposes an automatic two-way data transfer synchronization scheme that requires no involvement of host CPU in data transfer synchronization and offers real-time synchronization performance on subsystem side. ***(Abstract)***

Keywords-eMMC; subsystem; ECSS; DDR, ASRHA; ASR, synchronization; Input Buffer; Output Buffer

I. INTRODUCTION

The eMMC specification [1] is designed for mobile devices and is capable of storing data and code. It is intended to offer high data transfer bandwidth of 208MB/s when it runs at 104MHz frequency dual data rate (DDR) for large data transfers, while maintaining low power consumption. Advanced smart devices and handsets can use eMMC connected subsystems to expand computing power and content storage. In this paper, an eMMC Connected Subsystem (ECSS) is connected to a host system via eMMC interface and conducts a functional task by receiving input from host system and outputting processed results back to host system. Figure 1 demonstrates such an ECSS example of eMMC system that connects to Freescale iMX53 host system.

II. STATEMENT OF PROBLEM

ECSS applications range from high-end mobile devices that require high-definition video storage and extraction of advanced multimedia features, to embedded speech recognition systems that submit computationally intensive scoring tasks to ECSS for processing.

One of such applications is hardware accelerator for Automatic Speech Recognition (ASRHA) engine. Without loss of generality, in the following sections, we will use ASRHA as an example to illustrate the proposed scheme in detail. In ASRHA, there are two FIFO buffers, one is input buffer (IB) and the other is output buffer (OB). Host system writes speech data to IB on ASRHA based on the following two criteria:

- On host side, input data is generated by Automatic Speech Recognition (ASR) engine.
- On ASRHA side, IB has enough space to receive input data.

Similarly, host system retrieves processed data by ASRHA from OB based on:

- On host side, buffer for receiving output data is available.
- On ASRHA side, OB is not empty.

On iMX53 platform, the Enhanced Secured Digital Host Controller Version 3 (ESDHCV3) provides an interface between the host system and the ECSS [2], as depicted in Figure 1. The ESDHC acts as a bridge, passing host bus transactions to the device by sending commands and performing data accesses to/from ASRHA.

In ASRHA application, ASR engine off-loads most of its search job to ASRHA. In doing so, it generates input list that needs to be transferred to IB on ASRHA chip from host frame by frame. ASRHA accelerates the search phase of ASR by calculating HMM scores in addition to the distance scores by taking input list from ASR as input. As a result,

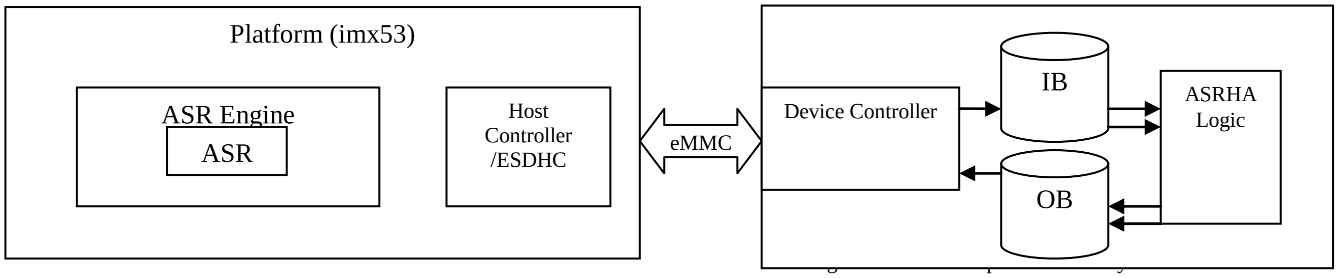

Figure 1. An Example of ECCS System

output list, created in the OB by ASRHA chip, is transferred to host for further processing via eMMC bus.

To facilitate real-time speech recognition, input list and output list have to be transferred with least latency in order to ensure fast decoding of speech signals. How to synchronize the data transfer between host system and ASRHA will directly impact data transfer rate and host CPU load. As shown in Figure 2, ASRHA system could hang when a deadlock situation occurs. This deadlock is caused by OB full in the midst of host writing to IB. ASRHA will not be able to unload and process IB data if the results from the processing have no place to store due to OB full.

Conventionally, the synchronization of data transfer is by polling, i.e., busy-and-wait approach. Specifically, the polling strategy can be explained below:

- Before the host initiates an eMMC command to send input list to ASRHA chip, it first polls the chip to find out the size of available input buffer using one eMMC command (for instance, CMD17 (read single block)). If the desired buffer size is not available, it has to poll again until the buffer is available.
- Similarly, when a host wants to read output list from ASRHA chip, it has to poll the chip to read the size of available output list. Multiple polls may be required before desired chunk size of output data is available. Here, chunk size is multiple times of blocks size.

Polling has been widely used in early computing system due to its simplicity. However, keeping polling will drain CPU power and lead to inefficiency in multitasking environments. Therefore, in recent systems interrupt-based notification mechanism has been used together with or replaced busy-wait approach on most occasions, especially for complex embedded systems for real time performance. For instance, in [3], a combined approach of polling and interrupt was investigated for message handling. Another side effect for ASRHA is, if a host doesn't poll ASRHA in time, ASRHA could either starve for input list or could stall because of OB full as shown in Figure 2.

For applications like ASRHA, however, interrupt service is not available to inform host of subsystem status changes regarding the usage situation of input buffer and output buffer. Therefore polling is necessary.

III. PROPOSED SCHEME

It is important to reduce data transfer overhead and improve effective data bandwidth so the host processor can be freed for other application tasks. To address the above mentioned issues, automatic data transfer synchronization has been proposed based on the following considerations:

- Utilize Ready/Busy mechanism provided by eMMC protocol to synchronize data transfer.
- Absorb data transfer synchronization overhead on ECCS side.
- Synchronize data transfer on ECSS side
- Eliminate polling from host system, thus reduce CPU involvement in eMMC data transfer as much as possible.

The proposal can be elaborated in the following two aspects, eMMC busy/ready control and device status return (R1) of eMMC commands.

A. eMMC Busy/Ready

The idea is instead of polling ASRHA for available buffer size or data size, host will send input list to ASRHA while ASRHA signals device BUSY by pulling D0 low when IB has no free space of input data. Only when the size of free IB for next transaction of IB data transfer meets a predefined threshold, it will remove BUSY signal and issue Ready signal. In this case, the host only submits ONE eMMC send multiple blocks command (CMD25) to ESDHC controller and it then goes to sleep to allow other tasks running until the arrival of controller notification when the command has been fully executed. This approach enables the host CPU to work on other critical tasks. More importantly, it prevents subsystem from hanging due to insufficient free IB buffer space when OB is full.

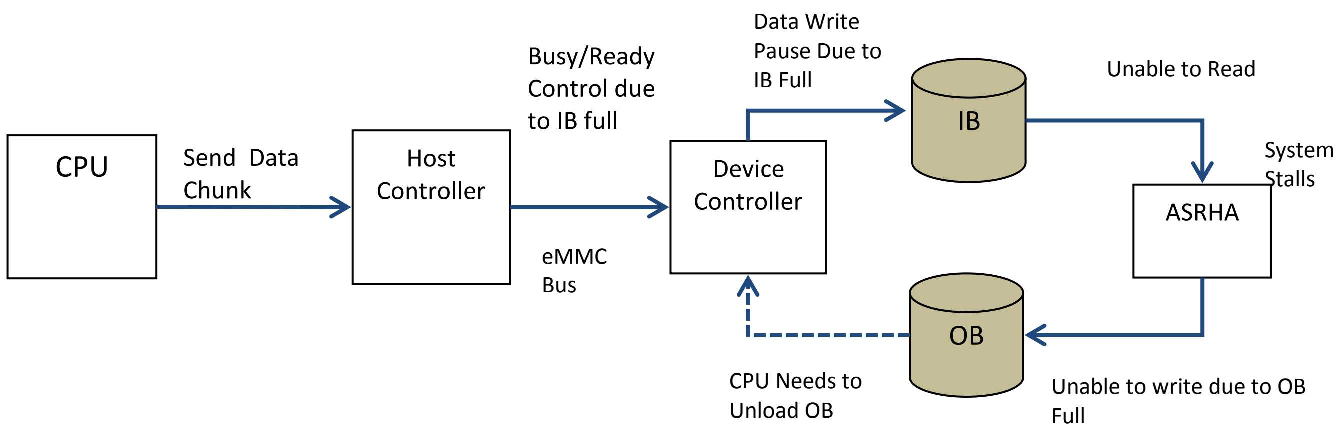

Figure 2. An Example of ECCS System

What happens to the internal activity of the controller? After receiving an eMMC data Read/Write request, ESDHC controller first checks whether ASRHA is busy or not. If it is busy, the controller will wait until it is ready unless timeout occurs. Therefore, when the ESDHC controller finds out ASRHA is ready for receiving data, it will then go ahead to transfer input list data to ASRHA.

B. eMMC Device Status

Based on JEDEC eMMC standard 4.4 [2], most data transfer commands, such as CMD 25 (Write Multiple Blocks), CMD12 (Stop Transmission), have R1 type response, as shown in Table 1. It means a 32-bit device status value will be returned to the host as the response to the eMMC command.

TABLE I. eMMC COMMAND RESPONSE TYPE

CMD	Description	Response
CMD 17	*Read Single Block*	*R1*
CMD 25	*Write Multiple Block*	*R1*
CMD 18	*Read Multiple Block*	*R1*
CMD 12	*Stop Transmission*	*R1*

Note that there are 8 reserved or unused bits in the status code. In addition, flash related error bits could be freed for application specific use as well. eMMC DDR rate data transfer always ends at Stop Transmission command (CMD12). Applications such as ASRHA can send back application status, for instance, OB data size, to host system using the status code.

IV. IMPLEMENTATION

The key points with Input Buffer write is ASRHA will guarantee free IB space available for host next batch write operation, at the end of each chunk write, by asserting and deserting "Busy" signal to synchronize with host. Since OB full will stall the activities of ASRHA, it should be prevented by sending Number of Available Output Blocks (NOB) via Device Status bits to host and host will give high priority to unload OB over writing to IB.

Similarly, ASRHA enables a host to read OB buffer one chunk at a time by asserting and deserting "Busy" signal to inform host whether it is ready or not.

A. ASRHA State Transition Table for IB Write

As shown in Table II, T1, the threshold for free space in IB and T2, the threshold for available data size in OB, are predetermined and can be tuned by applications.

ASRHA State 1 could transition to one of three different states after giving Busy signal as output. In this state, NOB is below watermark T2 and IB need increase free space for next incoming IB chunk by enabling Busy signal. In the middle of the increasing, NOB can grow as well and hit watermark T2. When either IB_free (free space in IB) growing enough or NOB hitting watermark, Busy can be turned off.

TABLE II. ASRHA STATE TRANSITION TABLE FOR IB WRITE

State #	Current State		Next State		Output
	IB_free > T1	*NOB > T2*	*IB_free> T1*	*NOB > T2*	*Busy*
1	*0*	*0*	*1*	*0*	*1->0*
	0	*0*	*0*	*1*	*1->0*
	0	*0*	*1*	*1*	*1->0*
2	*0*	*1*	*0*	*1*	*0*
3	*1*	*0*	*1*	*0*	*0*
4	*1*	*1*	*1*	*1*	*0*

B. ASRHA State Transition Table for OB Read

As shown in Table III, T1, the threshold for free space in IB and T2, T3, the thresholds for available data size in OB, are predetermined and can be tuned by applications.

During OB read, free space in IB still needs to be examined to ensure next transaction of IB write is safe. Therefore IB_free is part of current state for OB read besides NOB for OB.

TABLE III. ASRHA STATE TRANSITION TABLE FOR OB READ

State #	Current State			Next State			Output
	NOB < T3	*IB_free > T1*	*NOB >T2*	*NOB < T3*	*IB_free > T1*	*NOB >T2*	*Busy*
1	*0*	*0*	*0*	*0*	*1*	*0*	*1->0*
	0	*0*	*0*	*0*	*0*	*1*	*1->0*
	0	*0*	*0*	*0*	*1*	*1*	*1->0*
2	*0*	*1*	*x*	*0*	*1*	*x*	*0*
3	*0*	*0*	*1*	*0*	*0*	*0*	*0*
4	*1*	*x*	*x*	*0*	*x*	*x*	*1->0*

Table III lists current state and next state to transition to when outputting Busy as output. The next state will be same as current state if no Busy is set. At ASRHA State 1, NOB needs increase. However, before OB read finishes, we have to make sure IB has space for next input chunk. When this happens, sending input is not done yet. Therefore set Busy until IB_free is greater than a predefined value T1 or NOB hits watermark T2. State 2 and State 3 transition to the same states without setting Busy signal.

In ASRHA State 4, if NOB is below the watermark T2, Busy signal is activated to increase NOB until NOB < T3 is not true. When ASRHA is in this state, it indicates host finished sending input data and starts reading OB data.

V. EXPERIMENTAL RESULTS

The experiments are conducted on a Freescale iMX6 SABRE board running ARM® Cortex® A9 processor, which is connected to an Altera FPGA board that implements ASRHA logic. The host system is a speech recognition engine that outsources HMM search task to

ASRHA. The average data transfer rate is around 60MB/second with eMMC data transfer chunk size of 10KB.

In our experiments, the average polling occurrence is around 1.6 polls to each transferred input data chunk when using the conventional polling approach.

Our main concerns here are system CPU load and eMMC data transfer bandwidth [4-6].

First, let us look at CPU load. It shows 20% CPU load when polling is used to synchronize data transfer between CPU and ASRHA, as opposed to 16% CPU load using proposed Non-polling approach, which is 25% improvement.

Second, assuming 10% of the cases require extra polling and 10KB per chunk for eMMC data transfer, in a multi-tasking environment, delay is added between polling so other tasks can run as well in the meantime. Given 300us for the delay between polling, the proposed data transfer scheme speeds up data transfer rate by 18% compared to the polling approach.

VI. DISCUSSIONS

In the above sections, the proposed data transfer synchronization scheme has been discussed. The proposal is motivated by the idea of streamlining data read and write with least host CPU participation. The benefit can be summarized below:

First, the automatic two-way data transfer synchronization turns ECSS accesses into a simple reliable read/write operation. In other words, by hiding the complexity of ECSS technology, an easy interface to host system has been provided. This will enhance data transfer capability and shorten the development cycle and time to market.

Second, the host system workload has been reduced by outsourcing the data transfer synchronization task to ECSS. Least CPU involvement means higher system efficiency and faster responsiveness.

(Spansion®, the Spansion logo, and combinations thereof, are trademarks and registered trademarks of Spansion LLC in the United States and other countries. ARM and Cortex are registered trademarks of ARM Limited. Other names used are for informational purposes only and may be trademarks of their respective owner.)

ACKNOWLEDGMENT

The author would like to thank System Software Team at Spansion for making this publication possible.

REFERENCES

[1] JEDEC Solid State Technolgoy Assocation, "Jedec Standard, JESD84-A44, MMCA 4.4" Multimedia Card Assocation, March 2009.

[2] Eureka Technology, EP568 eMMC 4.5 Card Controller datasheet

[3] O. Maquelin, G. R. Gao, H. HJ Hum, K. B. Theobald, "Polling Watchdog: Combing Polling and Interrupts for Efficient Message Handling", ICSA's 96 Proceedings of the 23rd Annual Internal Symposium on Computer Architecture, May 1996.

[4] M. Loukides, "System Performance Tuning", O'Reilly & Assocates, Inc, 1991.

[5] H. Liu, "Software Performance and Scalability: A Quantitative Approach", John Wiley & Sons, Sep 20, 2011.

[6] A. S. Tanenbaum, "Operating Systems: Design and Implementation", Prentice-Hall, Inc., Upper Saddle River, NJ, 1987.