Development of the Embedded Multi Media Card Platform Based on FPGA

Songyan Liu(✉), Ting Chen, Shangru Wu, and Cheng Zhang

Electronic Engineering College, Heilongjiang University,
Xuefu Road 74, Harbin 150080, China
liusongyan@hlju.edu.cn,
{2151302,2141258,2161419}@s.hlju.edu.cn

Abstract. For the validation of eMMC device performance problems involving the effectiveness of testing and non-real time on parameters controlling, it may not be possible to obtain the performance data flexibly and efficiently, requiring consideration of the multi-channel parallel processing and real-time controlling. This paper presents a development platform for eMMC 5.0 device based on Zynq-7000. By combining hardware and software design, this platform is able to support eight eMMC devices working in parallel and get testing information in real time. Meanwhile, the device driver aims at achieving high performance data transfer by using DMA.

Keywords: eMMC · Zynq · Parallelism · DMA

1 Introduction

As the storage device widely used in mobile devices, this requires the device having the characteristics of small volume, big storage capacity, high data rate and short development cycle. Embedded Multi-Media Card (eMMC) consisted of the NAND Flash and a controller is perfect for these immediate needs. One of the major advantages of eMMC is that it provides the standardized interfaces to the external devices, which make it easier for developers to develop without dealing with the compatibility of NAND Flash. Facing with the fast-growing eMMC market, it is necessary to design an eMMC development platform, which is used to validate the stability, reliability, and veracity of the product during the development and testing stage.

In recent years, many domestic and foreign researchers have been proposed some development and testing solutions for flash memory. Kim et al. designed a development platform for flash memory solid state disks, which adopt a Xilinx Virtex-4 FPGA as the main processor [1]. The platform has four NAND Flash memory modules and supports different SSD architectures. Wei et al. presented a platform for NAND Flash based Zynq [2]. They combined the programmable logic with a processing system within Zynq to achieve sequence control, bad block management and error correction. Fu et al. proposed a test system for eMMC 5.0 devices based on FPGA [3]. They used Verilog hardware description language to implement the control of eMMC device. However, the system can only control an eMMC device to send commands at one time.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
X. Gu et al. (Eds.): MLICOM 2017, Part I, LNICST 226, pp. 648–656, 2018.
https://doi.org/10.1007/978-3-319-73564-1_66

Furthermore, there are some researches on the storage performance of eMMC device. Deng suggested two methods of automatic data transmission synchronization, with emphasis on eMMC busy/ready controlling and device status returning [4]. Amato et al. putted forward four eMMC key performance indicators: sequential read, sequential write, random read, and random write by analyzing the model of controller [5]. To address the random write performance issue, Byungjo Kim et al. introduced a way of the background command [6]. Compared with the conventional power-off way, the random writing capability has gone up by 173% in this method. The studies above are using different approaches to improve the efficiency of data reading/writing.

In order to validate the eMMC devices' operation and data reading/writing performance, this paper aims to design an eMMC platform based on the eMMC 5.0 protocol. The remainder of this paper is organized into five sections. Section 2 introduces the related background about eMMC. Section 3 describes the overall architecture of this platform. Section 4 details the design of software, and the result of the experiment is given in Sect. 5. The Final section concludes the paper.

2 Background

EMMC, a storage card oriented to smart phone and table computer, is made up of NAND Flash memory and a storage controller. The first version of embedded memory standard specification was released by the Joint Electron Device Engineering Council (JEDEC) in 2007. Today, it has already been updated to the eMMC 5.1 version. EMMC 5.0 supports three data transfer modes: 1-bit, 4-bit and 8-bit. Its maximum data transfer rate, 400 MB/s in HS400 mode, is the same with eMMC 5.1. So it is with good graces by manufacturers of mobile device.

All manipulation of eMMC device is based on the protocol. The eMMC system has five operation modes: boot mode, device identification mode, interrupt mode, data transfer mode and inactive mode. After power up, if the device received the CMD0 with argument of 0xF0F0F0F0, it would be set in the boot mode. Otherwise, it would go into device identification mode.

To realize eMMC bus data transmission, the host needs to some special signals, including command, response, and data. All the commands and responses are transferred on the CMD line. Each of them begins with a start bit and terminates with an end bit. The second bit indicated the transmission direction. If the bit is 1, it means this is a host command, otherwise card response. A 7-bit CRC checksum is used to guarantee the correctness of transmission.

Furthermore, if a data read-write command is sent, the host sends the data block on the DAT lines subsequently. If there are no data in these lines, the lines will hold a high level until data block is arrived. It is important to determine whether the mode is single block or multiple blocks before executing read and write command. Figure 1 shows read and write operations. The multiple block transmission can be terminated by the use of CMD12. A busy signal on the DAT0 line is used to indicate that the device is writing now. During the standby state, eMMC device could be switched into a sleep state to save the power consumption.

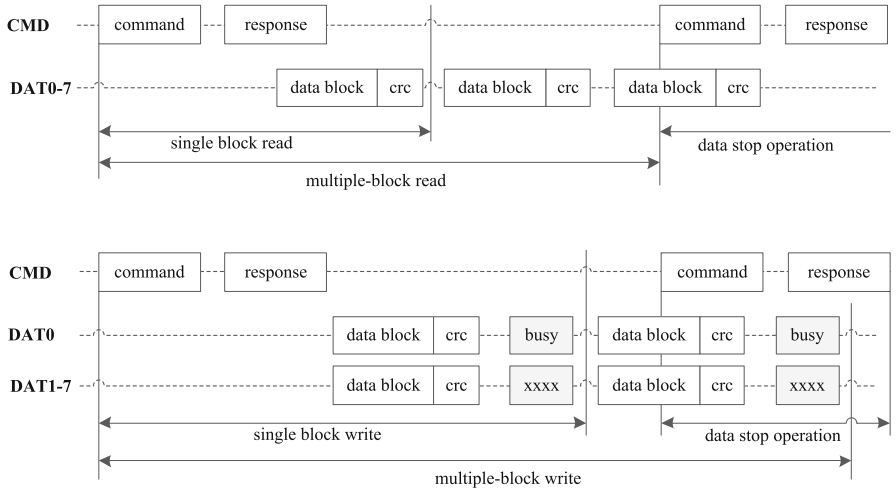

Fig. 1. The read and write operation.

3 System Design

The eMMC platform consists of three parts: PC client, server application and eMMC control module. The specific design of this platform is shown in Fig. 2.

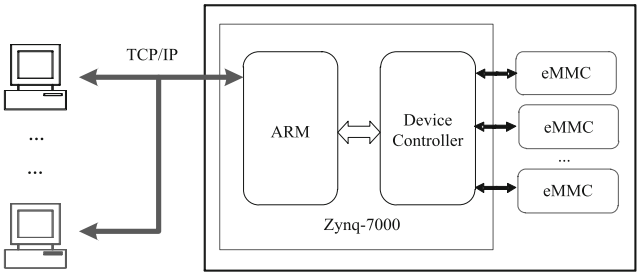

Fig. 2. The overview of eMMC development platform.

The hardware is composed of a Zynq-7000, eMMC chips, power module, network interface, USB 3.0 port and serial port. The Zynq-7000 chip is mainly composed of dual-core ARM9 processor and FPGA material. For this structure, not only the difficulty in data interaction between processor with FPGA can be solved, but also it reduces the system's power consumption and enhances productivity. It needs to accomplish the following work: (1) receive commands from the client; (2) control eMMC chips to complete operation task; (3) calculate the data performance; (4) send back the response to the client. The response will be returned after the command is

executed. In order to improve the data processing efficiency, this system is designed to support eight eMMCs working in parallel.

The client helps users to send commands, acquire response information and check the error status. If a fatal error occurs, the current processing will stop. This application is visual and concise so that it is very convenient to be used.

The server takes charge of establishing the connection with the clients. It runs on the PetaLinux operating system, which runs on the ARM9 processor. The PetaLinux, oriented for the MicroBlaze microprocessor soft cores, has a set of software development kit for Xilinx FPGAs. Not only does it provide the BSP's builder, but it also provides a lot of program templates to design the device driver and application. In this way, it can simplify the processing of system transplant and shorten the development cycle.

4 Implementation of Software

This section describes the software implementation of this platform in detail. Software architecture involves client program, server program and block driver. The device driver plays a key role in the overall system. Figure 3 depicts the software block diagram.

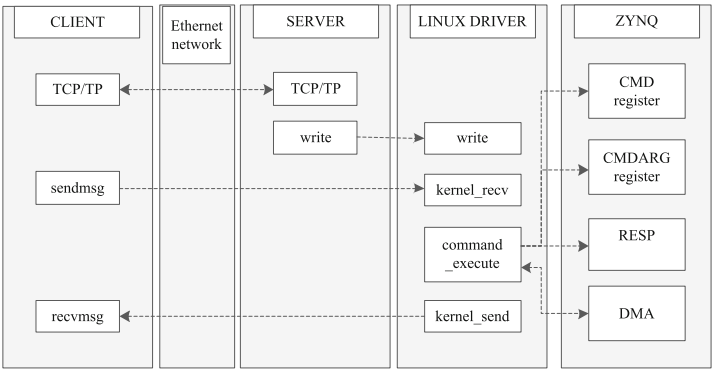

Fig. 3. The software block diagram.

4.1 PC Client

The main goals of this work are to packet the data exactly, establish reliable network connection and provide access to error checking. This design provides a user interface to input commands and arguments flexible. A lookup table is used to set clock frequency and determine the level of error. All of the error message will be printed to the user. If there is a fatal error, the current task would be interrupted, which achieves the efficiency of the system. This approach is quite convenient and accurately for a great number of commands.

Command Parsing. Since each client instruction contains some configuration parameters, including command index, command argument, data and block size, it is necessary to packet the instruction data at a specified format. Package data into entries is the main task of this module, which can be executed when the user inputs a series of parameters. Depending upon its parameter types, the order is parsed into various forms. If the command is a read and write command, this module should create the data buffer, which is used to hold the data file. The parsed commands are merged in a structure and then transferred to the device.

Message Scheduling. This module is responsible for the communication with the hardware devices to obtain the response information. Thus, it provides two main functions: (1) send and receive data; (2) check the returned message so that it can be monitored implementation of the commands and see whether any error occurred.

To make sure that all the data can be processed in real-time, this design applies a method: it communicates with the kernel layer directly. Considering the efficiency of this system, the command entries are transferred in a batch way. In other words, one or more commands can be transmitted to the device.

4.2 Interface of Application

As the kernel layer cannot establish a network connection over the TCP/IP protocol easily, it is required to set up the connection at the application level. Following this design, the server also has a feature that it could establish network connections with multi-user. As the number of clients request increases, the server's response rate may be slow commonly. To solve this problem, this design adopts the I/O multiplexing technology [7], which could be reduced the consumption of system resources. The idea of this mechanism is that it monitors the state of all socket descriptors. If there are any changes, the read event will be triggered [8].

4.3 Device Driver

In order to realize data transmission between ARM and eMMC devices, this research develops a device driver for the eMMC controller. The software flow diagram of the device driver is shown in Fig. 4. And the driver performs the following steps.

- Device initialization, requesting an interrupt for the eMMC device interrupt event, initializing the controller.
- Receiving commands from users.
- Once the command data have been successfully received, the command will be sent to eMMCs.
- If it read/write command, DMA operation would be started. Then creating send/write descriptors, loading descriptors, and initiating a DMA transfer.
- Returning the response to the users.

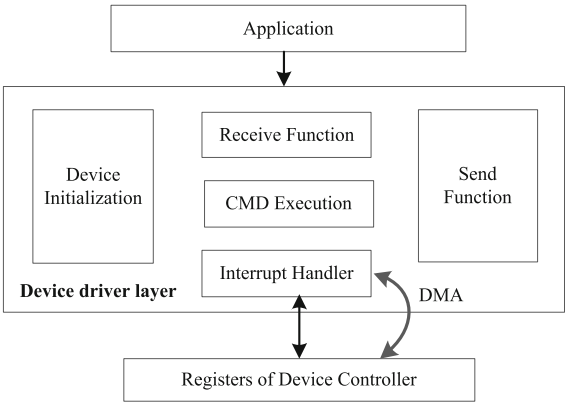

Fig. 4. The device driver software model.

Command Execution. The driver invokes the `emmct_send_serial_command()` function to complete the controlling of eMMC devices. Because the device contains specific information in different states, the implementation of eMMC command depends on the current device state. Given this fact, the state checking module was designed, which verifies that the next command operation conforms to the current state. On successful validation of the device status, the command is allowed to send to the device. The command execution has been elaborated in the following aspects:

- Disable interrupt before sending the command on the CMD line.
- Set the correct clock frequency according to current command.
- Initiate the command sending.
- Enable the interrupt.
- Check the error status, and then a command execution has completed.

Interrupt Handler. After the command or data has been sent to the eMMC device, the device generates an interrupt to the controller. Since the interrupt types are various, the driver should judge what interrupt is raised based on the value of mask interrupt status. To illustrate the interrupt has done, this interruption mechanism should make an interruption finished sign at appropriate times. Otherwise, it is considered as a timeout.

This interrupt will read the response from the response registers, and read pending data from the FIFO. Five types of eMMC response might be resulted, depending on the type of command. In addition to the R2, the length of other responses all is 48 bits. If a response error happens, this module might check the specific error message by reading the interrupt mask register. Figure 5 depicts the architecture of interrupt handler.

Data Read and Write. To transmit data on the DAT lines, two modes of data transfer can be chosen, single block data and multiple block data transfer. While, there are two types of multi-block transaction, open-ended multiple block read/write and multiple block read/write with pre-defined block count. It can change the block numbers by using CMD23 before the actual read/write command.

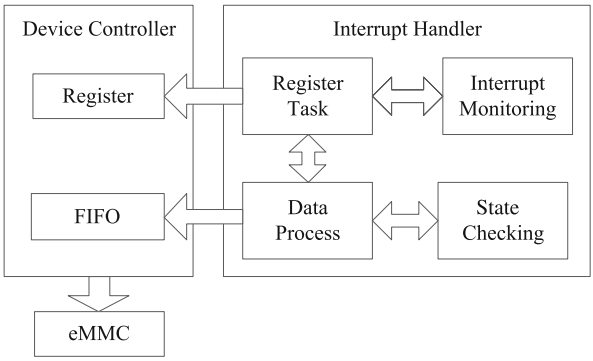

Fig. 5. Architecture of interrupt handler.

In order to improve the efficiency of this system, the DMA based on ring mode structure is used [9]. In this mode, each descriptor points to two different data buffers. The buffer1 holds a pointer to the data buffer, and buffer2 is not used in this driver. When the system needs to read/write data, the only thing it required is sending the command to eMMCs on each channel. The DMA engine then takes care of the operations of reading and writing without the processor's intervention. However, there are some undesired data stored in the DMA cache sometimes. If the DMA operation is initiated at this time, a data reading/writing error might be occurred. To address this issue, this design invokes the `dma_sync_single_for_device()` and `dma_sync_single_for_cpu()` functions to make the cached entry invalid and ensures the security of the buffer access.

5 Experiment and Result

To verify the feasibility and efficiency, we have been implemented a suite of functional tests for this system. The application connects to this platform via a network interface. And we choose eight eMMC 5.0 devices (8 GB) made by Skymedi as test objects. Five test routines are completed: initialization, bus test, single/multiple block write, single/multiple block read and erase. Table 1 shows the specific procedure of this experiment.

Table 1. The items of eMMC commands

Test item	Command sequence
Initialization	CMD0 -> CMD1 -> CMD2 -> CMD3 -> CMD9
Bus test	CMD19 -> CMD14
Single/multiple block write	CMD13 -> CMD7 -> CMD16 -> CMD13 -> CMD24/CMD25
Single/multiple block read	CMD13 -> CMD7 -> CMD16 -> CMD13 -> CMD17/CMD18
Erase	CMD13 -> CMD7 -> CMD35 -> CMD36 -> CMD38

For the data transfer performance, we can obtain the data transfer rate by calling `emmc_dev_data_perform()` function. The total amount of data written in is 1024 MB. After finishing writing, we read the data back again in this experiment. Table 2 shows the write and read performance of the eMMCs respectively by high speed SDR mode and HS200 mode.

Table 2. The data rate for writing and reading

Mode	Write	Read
High speed	46.45 MB/s	48.02 MB/s
HS200	91.39 MB/s	97.11 MB/s

On the performance side, the eMMC 5.0 device can support the maximum clock frequency (200 MHz) in the HS400 mode. However, the maximum clock frequency of this platform can only reach 100 MHz. When the clock frequency is set to 100 MHz, there is a phenomenon of high frequency harmonic. Within this problem we are still finding a solution.

6 Conclusion

This paper described a development platform for testing eMMC 5.0 devices efficiently and timely. Using the client/server architecture, this design is allowed to control and obtain the command information in real time. The approach of multi-channel parallel processing is applied that improves the system performance. The system only needs to send read/write command to eMMCs on each channel, DMA controller then takes charge of the operations of reading and writing. From the experiment, the feasibility of the design has been proven and it features the average write speed of 91.39 MB/s and read speed of 97.11 MB/s in the HS200 mode. However, the issue of clock frequency, which has the limits of data transfer rate, is needed to improve in the future work.

References

1. Kim, H., Nam, E.H., Choi, K.S., Seong, Y.J., Choi, J.Y., Min, S.L.: Development platforms for flash memory solid state disks. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 527–528. IEEE (2008)
2. Wei, D., Gong, Y., Qiao, L., Deng, L.: A hardware-software co-design experiments platform for NAND flash based on Zynq. In: 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–7. IEEE (2014)
3. Fu, N., Li, Y., Liu, B., Xu, H., Zhang, Y.: Realization of controlling eMMC 5.0 device based on FPGA for automatic test system. In: 2015 IEEE AUTOTESTCON, pp. 251–255. IEEE (2015)
4. Deng, S.: A new data transfer scheme for eMMC connected subsystems (2014)
5. Amato, P., Caraccio, D., Confalonieri, E., Sforzin, M.: An analytical model of eMMC key performance indicators. In: 2015 IEEE International Memory Workshop (IMW), pp. 1–4. IEEE (2015)

6. Reddy, A.K., Paramasivam, P., Vemula, P.B.: Mobile secure data protection using eMMC RPMB partition. In: 2015 International Conference on Computing and Network Communications (CoCoNet), pp. 946–950. IEEE (2015)
7. Kim, C., Lee, C.: Design of eMMC controller with multiple channels. In: 2016 International SoC Design Conference (ISOCC), pp. 317–318. IEEE (2016)
8. Ribeiro, I.L.B., Kimura, B.Y.L.: Enabling efficient communications with session multipathing. In: 2014 Brazilian Symposium on Computer Networks and Distributed Systems (SBRC), pp. 231–238. IEEE (2014)
9. Kavianipour, H., Muschter, S., Bohm, C.: High performance FPGA-based DMA interface for PCIe. IEEE Trans. Nucl. Sci. **61**, 745–749 (2014)