AVR®32771: USB High speed Device Mass storage on SD/MMC card with optional AES

Features

- **High Speed USB for high read and write speed**
- **Modular code simplifies maintenance and extensions**
- **Widely supported USB MSC interface**
- **Encrypted data for increased safety (optional)**

1 Introduction

This application note is a description of a USB Mass Storage device, using High Speed USB for communication and a SD/MMC-card for storage. By default the code is compiled to run at the EVK1104 reference design board with the AT32UC3A3. This document contains a high-level description of the source code following the application note, including description of the USB and SD/MMC stacks and performance measurements for different configurations.

EVK1104 A32UC3A3 evaluation kit

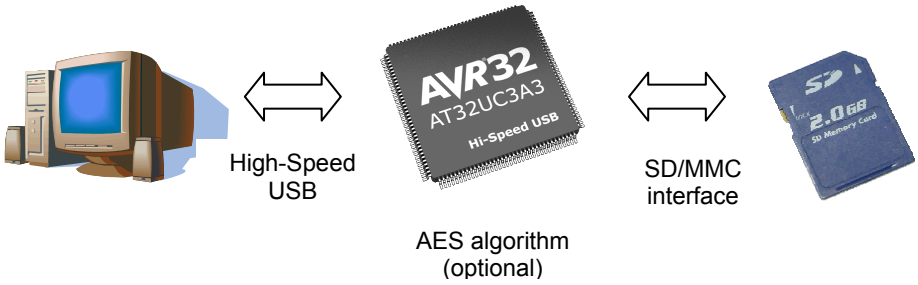
32-bit AVR® Microcontrollers

Application Note

Rev. 32132A-AVR32-02/10

2 Theory of Operation

In this application the AT32UC3A3 is set to communicate with two external devices, the host computer and a storage medium. In addition to these two communication interfaces, an AES algorithm can be utilized for encrypting the contents of the storage medium. High-speed USB is used for the communication between the host computer and the UC3, while a SD/MMC card is used as storage medium.



2.1 Transactions

All transactions, both reads and writes, are initiated by the host computer. They are sent as USB packages conforming to the USB Mass Storage Class (USB MSC) communication protocol. This is a standardized interface which is supported by natively by most, if not all, modern operating systems. The UC3 decodes the USB communication and translate it to the SD/MMC commands which are sent to the SD/MMC storage medium. If AES is turned on, the UC3 will encrypt all data before storing it in the storage medium. The SD/MMC-cards response, including any data read, will then be sent back to the UC3, and relayed to the host computer over USB.

2.2 Overhead

It is not possible to avoid introduction of some overhead due to the processing taking place in the UC3, but with clever driver design and heavy DMA-use this overhead is kept to a minimum. When relaying packages between the two busses as much as possible of the metadata is left unchanged, while the actual data is transferred solely by DMA. If AES is turned on, the hardware AES module in the UC3 is used for encryption and decryption, and here as well DMA is used to move the data between the different hardware modules.

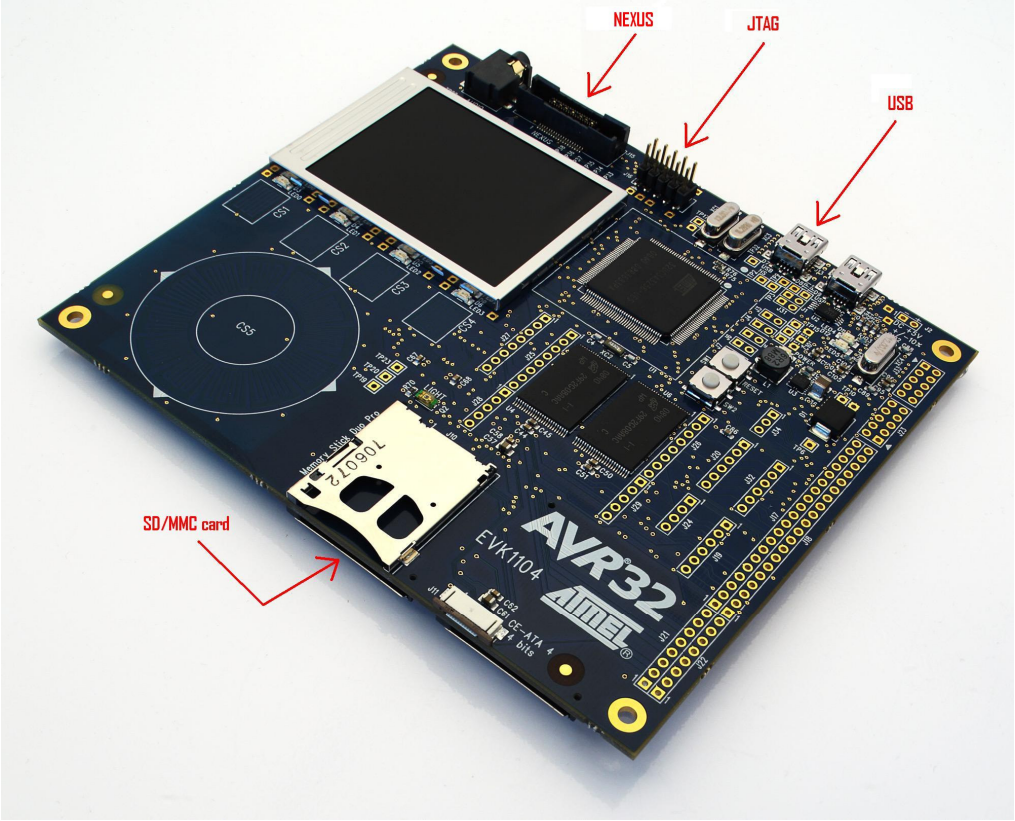
In total, the overhead and delays imposed when moving or processing the data is kept to a minimum. This combined with High-Speed USB allows the overall application fully utilize fast SD/MMC-cards and give very good read and write performance.

3 Using the application

The source code of this application and the drivers and library services it depends upon, are organized with performance, portability and ease of use in mind. This means that most of the complexity of the system is hidden inside the different drivers and services, simplifying application development. Those who are interested in learning more about the software architecture and the drivers can read about this in subsequent chapters.

3.1 Connecting the board

Before the EVK1104 can be programmed, it needs to be connected to the computer. A USB cable is needed for the application to function, as well as for powering the kit. You can choose between JTAG or NEXUS interface for programming and debugging. Lastly a SD/MMC-card is needed as storage medium for the application.



3.2 Building the application

In order to build the mass-storage application from the console, simply navigate to the application folder (*apps/mass-storage/*) and type *make.* This will produce an ELF file in the build folder (*build/mass-storage/atevk1104/).*

3.3 Programming the application

The application can be programmed to the UC3A3 flash either through the JTAG interface, or with a bootloader. In this application note we will only consider the former alternative. First you need to make sure your AVR-ONE or JTAGmkII is properly configured and connected to the EVK1104. Then run the command "*make program*" from the application folder. This will program the elf file of the application to the UC3A3 flash.

3.3.1 First time programming

The first time you are to program the application, you will need to go through some extra steps. If parts of the flash are locked, start with running a full chip erase. Then program in a trampoline which will jump from the reset vector to the start of the application. The use of a trampoline is mandatory for applications that are designed so that they can be used with bootloaders. If you are using a bootloader you do not need to program in the trampoline. Lastly, a USB serial number has to be programmed in. In commercial applications, this number must be unique for each device, but for demonstration purposes, just chose a random 12 digit hex number. Use the tool `set-serial.sh` found in the application folder for this task.

Example:

- Navigate to the apps folder (`apps/mass-storage/`)
- `make program`
- `./set-serial.sh 0123456789AB`
- Navigate to the build folder (`build/mass-storage/atevk1104/`)
- `avr32program program -evfinternal@0x80000000 trampoline.elf`
- Press the EVK1104 reset button to start the application

3.4 Debugging the application

AVR32 Studio can be used to examine how the application executes or debug it. Refer to Application Note AVR32769 to read more about how to set up AVR32 Studio to work with an external Makefile. The Makefile can be found in the apps folder (`apps/mass-storage/`).

3.5 Enabling AES encryption

WARNING: The AES functionality is currently in an experimental state and for developers use only. Reduced performance and stability is to be expected when this function is enabled.

The default configuration of this Application Note does not include any AES support. In order to compile in this support, some flags has to be set in the configuration file of the application. This will enable AES encryption for all data stored in the SD/MMC card, thus rendering it unreadable for unauthorized personnel.

Since the encryption is done on the block device level, even the filesystem is encrypted and an encrypted card will show up as unformatted if read without decrypting the data first. The same is the case when encryption is turned on; the SD/MMC card needs to be reformatted with encryption turned on, and all plain data on the card is invalidated.

3.5.1 Editing the application configuration

The mass storage applications configuration file (`apps/mass-storage/config-atevk1104.mk`) contains instructions to the build system, telling it what it needs to build and to some extent, how it should be built. Open this file for editing in your favourite editor, and scroll down to the following lines:

```
# Remove comment symbols to enable AES
#CONFIG_USE_AES=y
#CONFIG_AES=y
#CONFIG_BLKDEV_AES=y
```

Remove the comment symbols (#), in front of the three config lines. Save and close the file, then recompile and program the application (see section 3.3). When the application is restarted, your need to reformat your SD/MMC card with the encrypted filesystem, and you have an encrypted storage medium! Please note that the full space of the SD/MMC card is used for the encrypted filesystem, and all data previously on the card will be lost.

3.5.2 Setting the encryption key

In this Application Note we are using a static encryption key which is hardcoded into the application. It is set by the `cipher_key` variable in the main() function of the application (in `apps/mass-storage/main.c`). This is probably not how the encryption key will be set in a commercial application, but since key handling is very application specific, the simplest solution was chosen here. With the current implementation, the application needs to be recompiled in order to change the key. As when turning encryption on, this will require the SD/MMC card to be reformatted, and all data previously on the card is lost.

4 Software architecture

The software architecture used in this application note is inspired by how code is organized in the Linux® Kernel. Linux is a huge development project which has been able to combine rapid development cycles, a great number of developers, cutting edge technology and support for multiple hardware architectures with good code quality and stability. Some key points needed to achieve this are:

- Modular code
- Clean module interfaces
- Layered approach to driver development

The next sections will look into how this is reflected in the software architecture of this application note.

4.1 Pros and cons of modular code

Modular code consists of clearly defined modules, where each module takes care of a specific task or function. There is no hard line separating modular from non-modular code, rather some design principles that can be followed to a greater or lesser extent, of which the most important, simple module interfaces, will be discussed in the next section.

Modular code will often require a bit more work to actually write it, but users of the library services and drivers will certainly not notice this. Although a slight increase of code size or reduction of performance can sometimes be seen, the benefits gained from modular code heavily outweigh this. Increased modularity will result in increased reusability and testability of the code, reducing maintenance and making it easier to extend the code.

4.2 Module interfaces

The most important part of a module is how it interacts with the rest of the system, ie. its interface. The interface should be as simple as possible, making the module simple to use and hiding the complexity within from the rest of the system. This principle may make it more time consuming to develop the actual module, but this time is regained when developing the system around the module. In addition, simple module interfaces simplify testing of the modules and will thus assure the overall quality of the system.

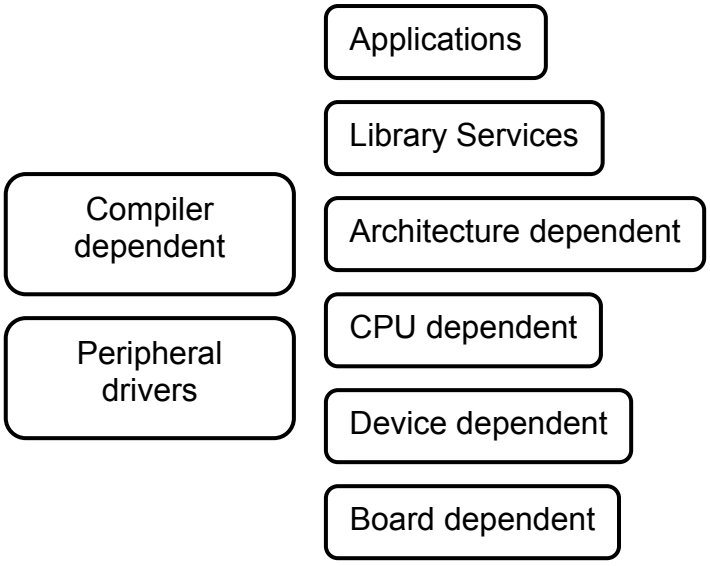
4.3 Layered drivers

A layered driver, or stack, is a driver where the functionality is grouped into several modules according to how close the functionality is to the actual hardware. This approach is particularly popular in large and complex drivers, like the USB driver.

In addition to giving more modular code, with the benefits described in the previous sections, the layered approach give drivers that are particularly easy to extend and port. New functionality is added by writing a new top-layer, for USB this layer will implement for instance the MSC or the HID class. Porting to new hardware is done by writing a new bottom level layer, often referred to as HAL (Hardware Abstraction Layer). Thus most of the code is left unchanged when adding top-level functionality or porting to new hardware, reducing development time and risk of introducing bugs considerably.

4.4 Code structure

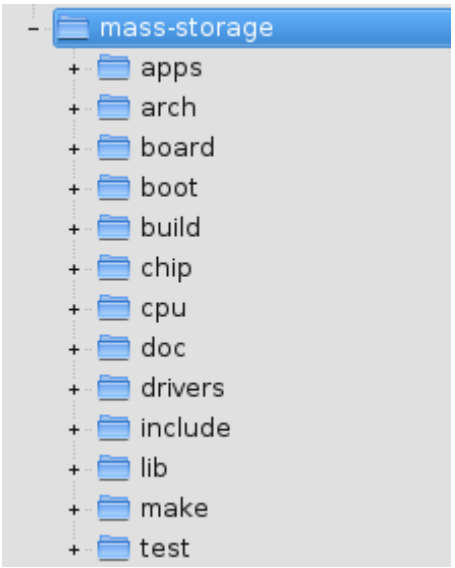
The source code in this application note is split into several categories and grouped by hardware dependence. This way as much code as possible is reused between the different architectures.



4.4.1 File organization

At the top level, the source code is organized in the following folders:

- apps – Applications
- arch – Architecture dependent code
- board – Board dependent code
- boot – Start-up code and trampoline
- build – Build folder
- chip – Device dependent code
- cpu – CPU dependent code
- doc – Code documentation
- drivers – Peripheral drivers
- include – Header files for drivers and library services
- lib – Library services
- make – The central part of the build-system
- test – Scripts for USB tests

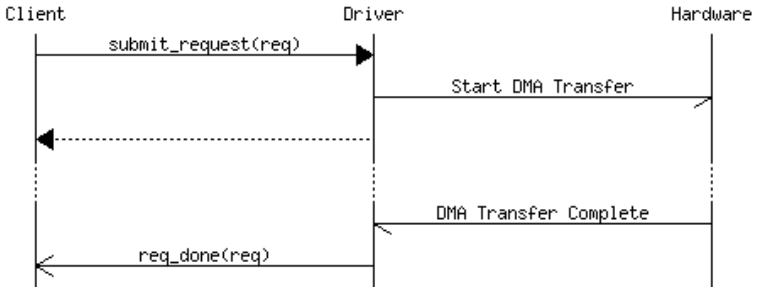


5 Driver design

The design strategy for the drivers in this application note has been to write optimal code that is easy to use. Thus most of the complexity in the drivers is hidden from the application itself. In order to free up system resources, all drivers have asynchronous interfaces and depend as much as possible on DMA.

5.1 Asynchronous drivers

An asynchronous driver will return control to the caller just after any transfer request has been submitted. It will not block until the transfer has been completed. This behavior allows the caller to perform other tasks while waiting for the driver to complete its. When the transaction as been completed, or failed for some reason, the driver will notify the calling application about this.



The downsides of asynchronous drivers are more complex internal structure, slightly more overhead and bigger flash and memory footprint. On most devices, these disadvantages are heavily outweighed by the advantages of using DMA and enabling multitasking in the applications.

5.2 The request structure

The interface of an asynchronous driver revolves around the request structure. This is a struct passed to the driver by the caller, containing a full description of the task the driver should perform. When the task is completed, the struct is updated and returned to the caller. A typical request struct will look something like this:

```
struct slits         buf_list
struct slist_node    node
unsigned long        flags
void                 (*req_done)(struct udc *udc,
                                 struct usb_request *req)
void                 *context
int                  status
size_t               bytes_xfered
```

The above struct is the USB request struct used in this application note. It contains the following elements:

- A list of data buffers for transmitting from or receiving into. Each buffer is a continuous region of memory, while the list could represent a segmented memory region.
- A list node used for queuing the request. Read more about request queues in the next section.

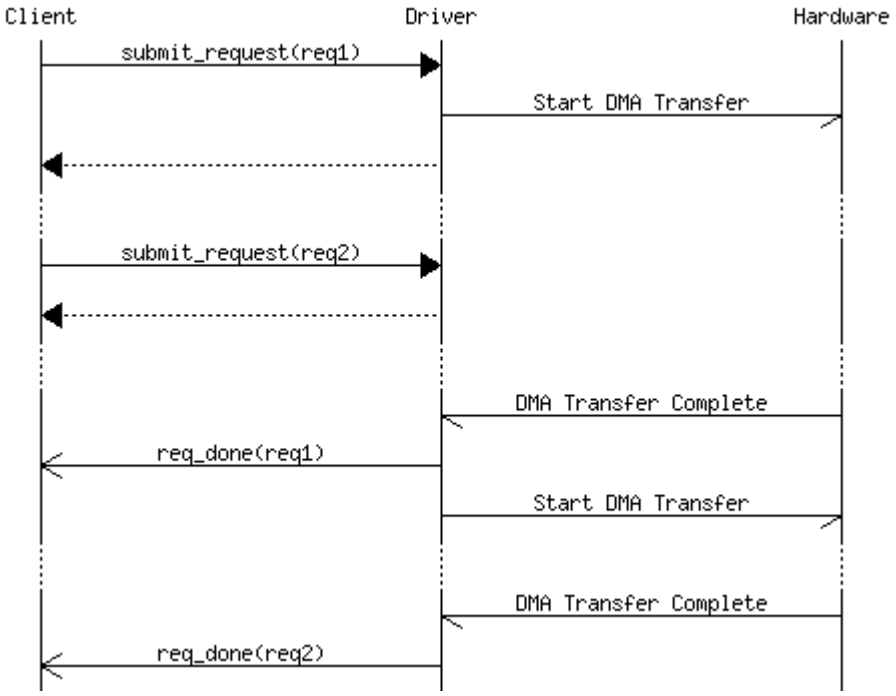
- Flags used to describe the request type and how the driver should handle it.
- A pointer to a callback function. This function is called by the driver when the request is completed. Pointers to the driver instance and the request itself are passed to the callback function.
- A pointer to arbitrary context data. This is typically used by the caller to convey its own state to the callback function. The driver does not use or modify this data.
- A status field set by the driver before calling the callback. This field will typically say that the request was completed successfully or, if errors were encountered, how the transfer failed.
- The last element is a count of the total number of bytes transferred. If the transaction failed or got aborted, this field will tell the caller how much of the submitted data that actually got transferred.

Most other requests contain the same or similar elements as the USB request analyzed here.

5.3 Queuing of requests

When an application can continue its execution after initiating a transaction, it will sometimes want to access the same peripheral again before the first transaction has completed. Few peripherals support this, and blocking the second request until the first has finished means the driver is only semi-asynchronous. The caller could handle this itself, by waiting for the request done callback for the first request, but this counteracts the principle of hiding the complexity inside the driver.

The solution is to enable the driver to queue up multiple requests. The caller can now submit several requests in close sequence or even simultaneously, without having to check if the underlying peripheral is ready or not. By utilizing a generic list framework and queuing the list node available in the request struct, the overhead in the driver is minimal.



5.4 Moving data

A key aspect of the mass-storage application is to quickly move large amounts of data between different peripherals. With both the USB and SD/MMC driver depending on DMA, the actually data shoveling is already taken care of. What remains is exchange of metadata, telling the drivers what data to find where, with as little overhead as possible. This is achieved by utilizing the same buffer structure throughout the system. The USB driver will simply hand over its list of buffers to the SD/MMC driver, or vice versa. This makes for very clean and low overhead inter-driver communication.

5.5 The block device

To further generalize the interface between the USB MSC and SD/MMC drivers, a generic block device is utilized. The block device is a simple interface that allows blocks of data to be read or written to addressable blocks on a underlying storage medium.

The SD/MMC driver implements this interface, eliminating the need for the USB MSC driver to know anything about the SD/MMC driver. By adding this extra layer to the communication, the USB MSC driver is simplified and kept generic, making it easy to re-use the same driver with a different storage medium.

6 USB driver

AT32UC3A3 is the first device in the UC3 series with High Speed USB capabilities. With rates up to 480 Mhz, this makes it the perfect device for applications demanding high data throughput. This chapter is intended to provide a brief overview of the USB driver, for more through documentation, please refer to the Doxygen generated source code documentation.

6.1 The USB protocol

USB is a widely used standard for communication between electronic devices, especially communication between desktop and laptop computers and devices connected to them. The protocol includes standards for a wide variety of communication types, including the mass storage class that are being using in this application note. This class is used to provide a generic interface to everything from external hard drives to SD-cards. In some situations it can make sense to use USB as the communication protocol internally in a computer, taking advantage of the generic interface with all its built-in functionality.

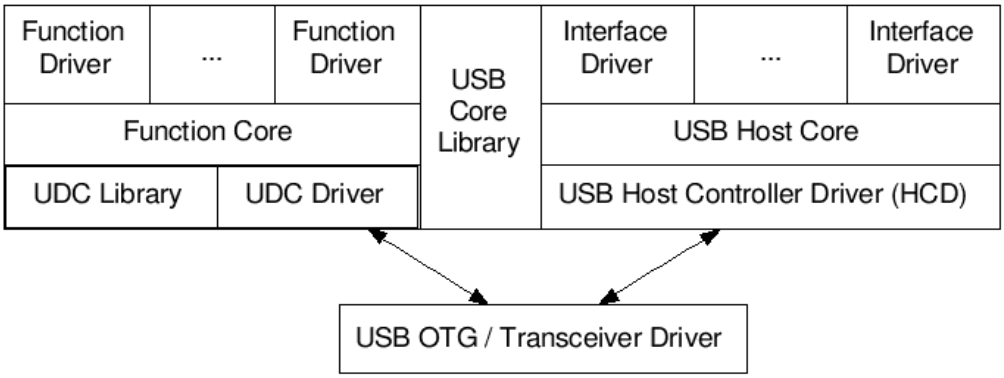
6.1.1 Compliancy to the USB standard

The USB protocol is no doubt one of the most complex communication protocols supported by embedded devices. Still, devices from a vast amount of producers work together remarkably well. An important reason for this is the thorough specification of all elements of the protocol provided by the USB Implementers Forum, combined with certification routines that make sure USB communication interfaces are implemented according to the specification.

In order to ensure that this Application Note conforms to the USB standard, both the USB High-Speed Peripheral and USB MSC tests are run on a regular basis.

6.2 Driver layers

The USB driver consists of several modules in a layered structure, with simple interfaces and low interconnection between the modules.



6.2.1 USB Device Controller driver

The UDC driver provides a low-level abstraction of the hardware. It is set up to handle interrupts generated by the underlying hardware and communicates these events to above layers. When necessary, it will call into the UDC library or function driver in order to process data or requests from the USB host.

6.2.2 USB Device Controller library

The UDC library provides helper functions for the UDC driver. Among other things, it will handle SETUP requests received on endpoint 0. UDC driver is set up to call the appropriate functions in the UDC library when such a request is received.

6.2.3 USB function driver core

As its name states, this is the core of the USB device driver. It keeps track of configurations, interfaces and settings and changes them or switches between them based on requests received from the host. Each configuration can have one or more interfaces, and these are all active when the configuration is. In addition the interfaces can have alternative settings that can be switched between at the host's discretion.

6.3 Mass storage class

The USB Mass Storage device class (USB MSC) is the protocol commonly used when storing data in USB devices. This protocol provides the USB host with a recognizable interface to the storage medium, making it easy for the host to use different types of flash or hard drives without having to know anything about the actual storage medium.

6.3.1 Serial number

The USB MSC specifies that every device should be identified by a unique serial number when enumerating. This number must be programmed in separately of the application during production. See section 3.3.1 for details on this.

6.3.2 Enumeration and host side driver

When the device is connected to the host, it will be identified as a USB Mass Storage device, enabling the host to use a generic driver for communication. A specialized driver is not needed in order to use the device. Once the enumeration process is completed, the device is ready for use.

6.4 Hardware design guidelines

High Speed USB communicates at 480 Mhz, which often is much higher than any other signals in the system. The high frequency requires some care to be taken when designing the hardware. Refer to design guidelines (available from usb.org) or reference designs for more details.

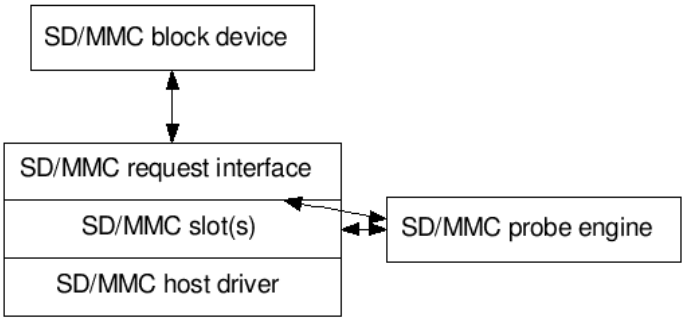
7 SD/MMC block device

The second part of the application is the SD/MMC driver stack. This stack provides a generic asynchronous block device interface to the USB MSC protocol.

7.1 Driver organization

The driver is organized in several layers, with the hardware specific host driver at the bottom, and a generic block device presentation at the top. In addition, a probe engine is running on the side, probing for card events. This will typically run in a workqueue, perhaps with a certain delay between runs. If a card has been inserted or removed since the previous run of the probe engine, it will alert the driver, which will take appropriate action.

Figure 7-1. SD/MMC driver organization



7.2 Cards supported

The driver should support all MMC, SD and SDHC cards, and has been successfully tested with the following cards:

- SanDisk® Ultra II
- SanDisk Extreme III, SDHC
- Kingston® SDHC
- Kingston Ultimate SDHC

8 Performance

Performance, measured as read and write speed, is a key aspect of any mass storage application. In this chapter we will look into what affects the performance, and what performance measurements to expect from this mass storage implementation.

8.1 USB and SD/MMC transfer speed

The performance of a mass storage application can be limited by both the USB protocol and the SD/MMC interface. High speed USB have a theoretical limit of 480Mbit/s, while the SD/MMC interface typically run at 33Mhz or 50Mhz, which give 132Mbit/s or 200Mbit/s with a 4-bit wide bus. Both these interfaces will experience some overhead, but unless there is a lot of other traffic on the USB, the SD/MMC interface will usually be the bottleneck.

8.2 Overhead

There are several sources of overhead in a mass storage application based on SD/MMC. Some are outside our control, but will be mentioned to give a better understanding on what performance to expect from a certain system.

8.2.1 USB and MMC protocols

In order to send information over either USB or the SD/MMC protocol, it needs to be packaged according to these protocols. Both the packaging and the handshaking needed to transmit a package will add to the overhead of the overall system. Especially when transferring small batches of data, this overhead can be very large. However, this overhead component will be seen on all mass storage applications, and can not be controlled by the implementation.

8.2.2 USB host and SD/MMC card

There are four parties involved in either a read or write operation. All operations are initiated by a USB host and sent to the USB device controller on the AVR32, which dispatches these as SD/MMC requests, sent by the AVR32 SD/MMC controller to the SD/MMC-card. Delays and overhead introduced by the USB host and SD/MMC-card are obviously independent on the mass storage implementation. Especially the SD/MMC-card will typically generate big delays, due to read access latency when reading and programming latency when writing. These are by far the biggest delays in the system, and will vary a great deal between different brands and models of SD/MMC-cards.

The programming latency is not only long, it is also unpredictable. This is because the SD/MMC-card may have to erase sectors before writing, or move data in order to level wear. In an effort to average out these effects and get reproducible figures, large amounts of data will be used when measuring write performance.

8.2.3 The mass storage application

The overhead and delays caused by the mass storage application is something that directly depends on the implementation chosen, and are the only aspects of the overall performance that can be controlled. Overhead can arise from handling handshaking, buffering data, interpreting and formatting data packages. Much of the overhead is fixed for each package of data, and will be most noticeable when transferring small data packages.

8.3 Results

Both read and write speed depends on the data package size, as well as the USB host and the SD/MMC-card. In order to reproduce a certain result it is important to recreate a similar environment. Package size may vary a lot between different situations, and is largely determined by the file system used and the files stored on the SD/MMC-card. Results here are posted for different packages sizes, and if the package mix of a certain system is known, the average performance can be estimated from these figures.

The benchmarks are performed with two different USB host chipsets and with Linux and Windows operating system on the host. In Linux the speed is determined by writing or reading large amounts of data from an unformatted card with the command *dd*. In Windows the application *ATTO benchmark* is used to determine the speed. This application creates a big file on a NTFS formatted card and then performs long sequential reads and writes on that file.

The read and write speed figures given below provide information about what to expect from a certain configuration.

8.3.1 Performance on Windows XP® host with ICH7 USB chipset

Table 8-1. Results

SD card	4kb (read, write)	64kb (read, write)	512kb (read, write)
Kingston SDHC	2.5MB/s, 0.5MB/s	11.8MB/s, 4.9MB/s	N/A, N/A
Kingston Ultimate	3.3MB/s, 1.0MB/s	12.2MB/s, 7.5MB/s	N/A, N/A
SanDisk Extreme III	2.8MB/s, 1.7MB/s	11.9MB/s, 8.8MB/s	N/A, N/A
Average	**2.9MB/s, 1.1MB/s**	**12.0MB/s, 7.1MB/s**	**N/A, N/A**

8.3.2 Performance on Windows® Server 2008 host with ICH8 USB chipset

Table 4-2. Results

SD card	4kb (read, write)	64kb (read, write)	512kb (read, write)
Kingston SDHC	3.6MB/s, 0.9MB/s	13.0MB/s, 7.6MB/s	N/A, N/A
Kingston Ultimate	4.4MB/s, 1.5MB/s	13.1MB/s, 7.7MB/s	N/A, N/A
SanDisk Extreme III	4.1MB/s, 2.2MB/s	12.8MB/s, 9.6MB/s	N/A, N/A
Average	**4.0MB/s, 1.5MB/s**	**13.0MB/s, 8.3MB/s**	**N/A, N/A**

8.3.3 Performance on Linux host with ICH7 USB chipset

Table 8-3. Results

SD card	4kb (read, write)	64kb (read, write)	512kb (read, write)
Kingston SDHC	3.8MB/s, 0.8MB/s	12.4MB/s, 7.7MB/s	15.0MB/s, 9.3MB/s
Kingston Ultimate	4.1MB/s, 1.4MB/s	12.7MB/s, 9.8MB/s	15.0MB/s, 12.6MB/s
SanDisk Extreme III	3.7MB/s, 2.1MB/s	12.4MB/s, 10.6MB/s	15.0MB/s, 13.1MB/s
Average	**3.9MB/s, 1.4MB/s**	**12.5MB/s, 9.4MB/s**	**15.0MB/s, 11.7MB/s**

8.4 Code size

Compiling the application without debugging console and asserts, gives the following code size with O2 optimization and gcc version 4.3.2:

```
   text    data     bss     dec     hex filename
  30880     192     992   32064    7d40 mass-storage.elf
```

9 References

1. Doxygen generated source code documentation
2. AVR32787: AVR32 UC3A3 High Speed USB Design Guidelines
3. AVR32769: How to Compile standalone AVR32 Software Framework in AVR32 Studio V2
4. EVK1104 Schematics
5. Universal Serial Bus webpage (http://www.usb.org)
6. SD-card overview (http://en.wikipedia.org/wiki/Secure_Digital_card)

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
Avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature