Application Report
SPRA881A – June 2003

A DSP/BIOS MMC Device Driver for TMS320C5509 DSP

Software Development Systems

ABSTRACT

This document describes the usage and design of the generic TMS320C5509 multimedia card (MMC) device driver. This device driver is written in conformance to the DSP/BIOS IOM device driver model and handles communication to and from the MMC. It uses the MMC controller on the DSP to access the card. It can be used as a general-purpose stand-alone mini-driver to access a Multimedia card or can be used by as a media driver by a file system driver.

Features:

- Non-blocking input/output mechanism.
- Multiple channels can be created for the same physical MMC device.
- Provides facility to abort or flush operations on a particular channel.
- The MMC driver can be specified using the configuration structures.
- Designed for (but not limited to) use with the file system drivers.

Contents

List of Figures

List of Tables

1 Usage

The device driver described here is an IOM mini-driver. That is, it is implemented as the lower layer of a 2-layer device driver model. The upper is typically, but not limited to, the DSP/BIOS GIO class driver that provides the application a common interface for I/O requests. For more information about the IOM device driver model as well as the GIO and other DSP/BIOS class drivers, see the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

This MMC media driver can be used as a media driver to interface with the MMC peripheral by the file system drivers on TMS320C5509 chips. However, it can also be used as a general-purpose stand-alone mini-driver to interface with the MMC. Figure 1 shows the data flow hierarchy between the components in a system when using the MMC driver as a media driver by the file system.

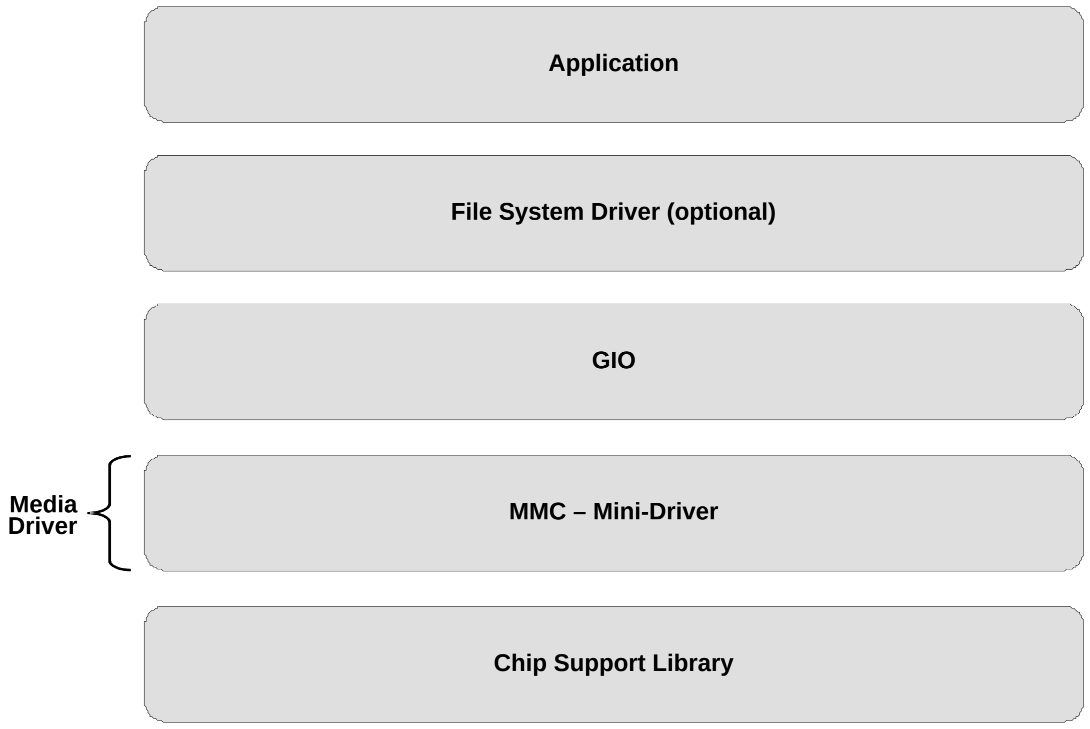

Figure 1. Media Device Driver Usage

1.1 Configuration

To use this device driver, a device entry must be added to the DSP/BIOS configuration tool for the MMC in user defined device drivers. The properties to be set for the device entry are described below:

- **Init function table:** Set to C5509_MMC_init.
- **Function table ptr:** Set to C5509_MMC_FXNS.
- **Function table type:** Select IOM_Fxns.
- **Device ID:** Set this to 0x0 or 0x1, depending on the MMC controller being used. If there is only one instance of an on-chip MMC controller, set this to 0x0. Device ID of 0x0 is used for the controller with base address at 0x4800, and Device ID of 0x1 is used for the controller with base address at 0x4C00.

- **Device params ptr:** A pointer to the device parameter structure (described in the following section). This property should never be set to 0x0, since the driver does not set any default parameters. This is a pointer to an object of type C5509_MMC_DevParams as defined in the header file c5509_mmc.h.
- **Device global data ptr:** A pointer to the device global handle structure. This is a pointer to an object of type C5509_MMC_DevObj as defined in the header file c5509_mmc.h

1.2 Device Parameters

A pointer to this structure should be passed when a device entry is made in the configuration tool.

```
typedef struct _C5509_MMC_DevParams
{
    Int                  versionId;
    MMC_InitObj          devParams;
    C5509_MMC_DrvrParams drvrParams;
    Uns                  ier0Mask
    Uns                  ier1Mask
} C5509_MMC_DevParams;


where, MMC_InitObj is defined in csl_mmcdat.h as follows:

typedef struct {
  Uint16 dmaEnable;
  Uint16 dat3EdgeDetection;
  Uint16 goIdle;
  Uint16 enableClkPin;
  Uint32 fdiv;
  Uint32 cdiv;
  Uint16 rspTimeout;
  Uint16 dataTimeout;
  Uint16 blockLen;
} MMC_SetupNative, MMC_InitObj;
```

C5509_MMC_DrvrParams is defined as:

```
typedef struct _C5509_MMC_DrvrParams
{
    Int mmcCardId;
    Int noChan;
    Int byteOrder;
    Int mmcDmaChR;
    Int mmcDmaChWR;
    Int *bufR;
    Int *bufWR;
    struct _C5509_MMC_ChanObj *pChan;
} C5509_MMC_DrvrParams;
```

Fields of C5509_MMC_DevParams:

- **versionId:** Version number of the driver.
- **devParams:** These are MMC specific configuration and operation parameters.
- **drvrParams:** These are driver specific configuration parameters and would have nothing specific to the MMC device.
- **ier0Mask:** Interrupt enable register 0 mask, set in the ISR.
- **ier1Mask:** Interrupt enable register 1 mask, set in the ISR.

Fields of MMC_InitObj:

- **dmaEnable:** This bit enables/disables DMA mode for data read and write. Currently the driver works only in non-DMA mode.
- **dat3EdgeDetection:** This field is reserved for internal use.
- **goIdle:** This parameter specifies whether the MMC should go to idle state when IDLE instruction is given or not. This should always be set to 0.
- **enableClkPin:** This field is reserved for internal use.
- **fdiv:** This parameter specifies the divide down factor of processor clock to MMC function clock.
- **cdiv:** This parameter specifies the divide down factor of MMC function clock to memory clock.
- **rspTimeout:** This parameter specifies the number of memory clocks to wait before reporting a response timeout once a command is issued.
- **dataTimeOut:** This parameter specifies the number of memory clocks to wait before reporting a data timeout.
- **blockLen:** This parameter specifies the block size (in bytes) for read/write operations. This should be set to the size of the sector on the media.

Fields of C5509_MMC_DrvrParams:

- **mmcCardId:** This is the relative card address (RCA) set by the application. The RCA value set for an MMC should be a positive value greater than zero.
- **noChan:** This parameter specifies the maximum number of channels required for this device.
- **byteOrder:** This parameter specifies the order in which bytes should be retrieved. It can take the values shown in Table 1.

Table 1. Byte Order Values

Value	Description
MEDIA_BYTESWAP	Swap the bytes of each word before returning it to the application.
MEDIA_BYTENORMAL	Don't do any swapping of bytes.

NOTE: These are defined in media.h.

- **mmcDmaChR:** This parameter specifies the DMA channel number for read operation. This will be unused for current non–DMA mode.
- **mmcDmaChWR:** This parameter specifies the DMA channel number for write operation. This will be unused for current non-DMA mode.
- **bufR:** This provides a buffer for DMA read (unused in non-DMA mode).
- **bufWR:** This provides a buffer for DMA write (unused in non-DMA mode).
- **pChan:** This is a pointer to an array of channel objects (noChan number of channel objects).
- **bufR:** This provides a buffer for DMA read (unused in non-DMA mode).
- **bufWR:** This provides a buffer for DMA write (unused in non-DMA mode).
- **pChan:** This is a pointer to an array of channel objects (noChan number of channel objects).

1.2.1 Device Global Object

The MMC driver internally maintains the state information in a device context. It uses the C5509_MMC_DevObj structure. The structure is as shown below.

```
typedef struct _C5509_MMC_DevObj {
    Int mmcDevNum;
    C5509_MMC_DrvrParams drvrParams;
    MdUns blockLen;
    QUE_Obj pendList;
    IOM_Packet *currPacket;
    MMC_CardObj cslCards;
    MMC_Handle cslMMC;
    MdUns buf[C5509_MMC_SECLEN];
    Int chOpenCount;
} C5509_MMC_DevObj;
```

- **mmcDevNum:** This stores an integer value indicating the base address of the MMC that is being used.
  - 0 for DEVICE0 (0x4800)
  - 1 for DEVICE1 (0x4C00)
- **blockLen:** This stores the block size (in bytes) for read/write operations.
- **pendList:** This is used to maintain a queue of pending jobs.
- **currPacket:** This stores a pointer to the current packet being processed.
- **cslCards:** MMC card object obtained from chip support library during initialization.
- **cslMMC:** MMC handle obtained from chip support library after opening.
- **buf[C5509_MMC_SECLEN]:** Buffer reserved for use by mini-driver. Sector length (C5509_MMC_SECLEN) is 256 words.

- **chOpenCount:** Number of logical channels open on the MMC device that this device object represents.

2 Architecture

This section provides the architectural details of the MMC mini-driver. It describes the design and implementation of the device driver. The various data structures used are also explained along with internal queue management. The driver uses various DSP/BIOS and CSL modules (see Appendix A).

2.1 Data Structures

This device driver uses two data structures to maintain its internal state. The data that is common across all instances, of an instance of an MMC controller, is maintained in a global state structure (C5509_MMC_DevObj) that is described in the previous sections. The data for each channel that is opened on the device is maintained in a channel object. This channel object structure is discussed in this section. This section also describes the structure used by the driver to maintain the request information for read/write operations.

2.1.1 Channel Object

The MMC driver internally maintains the channel context and state information using the C5509_MMC_ChanObj structure. A pointer to this structure is returned as a handle during channel creation operation.

```
typedef struct _C5509_MMC_ChanObj  {
    Int  mode;
    Int error;
    Bool inUse;
    IOM_TiomCallback  callback;
    Ptr callbackArg;
    C5509_MMC_DevObj  *pHandle;
} C5509_MMC_ChanObj;
```

- **mode:** This field indicates the mode in which the device has been opened. This value will always be IOM_INOUT for the MMC driver.
- **error:** This holds the error status for this channel.
- **inUse:** This maintains the usage status of the channel object, indicating whether it is in use or free.
- **callback:** This is the callback routine supplied by the GIO class driver.
- **callbackArg:** This field holds the callback arguments to be passed to the GIO class driver while invoking the call back .
- **pHandle:** Pointer to the global device handle.

2.1.2 Read/Write Structure

The media read write structure is used to store the information of a request made by the application.

```
typedef struct _MEDIA_RdWrObj
{
    Ptr    buf;
    LgUns  address;
    LgUns  subAddress;
    LgUns  length;
    Uns    reserved[24];
} MEDIA_RdWrObj;
```

- **buf:** This contains a pointer to the buffer specified by the application to be used for performing the read/write operations. This must point to a word-aligned buffer.
- **address:** This field indicates the sector from which to read from the MMC media.
- **subAddress:** The offset position in the sector specified in the SectorNumber.
- **length:** The number of words to be read. It should not be more than the number of words in a sector.
- **reserved:** This area is reserved for use by the driver.

2.2 MMC Mini-Driver Functionality

The MMC driver implements the interfaces required by the DSP/BIOS IOM driver model. As per the IOM mini- driver specification, it implements functions to bind device, create and delete channel, submit and control operations.

2.2.1 Device Binding

Device binding is done during initialization of the system. This performs the required initialization and sets up the MMC to an operational mode. The maximum number of channels required to be opened simultaneously is passed to this call. The memory for all the channel objects is also passed to the driver in the bind parameters.

2.2.2 Channel Creation

This function will be used by the class driver to create a channel and use it for read/write operations. The MMC driver should always be opened in input/output mode.

The driver returns a pointer to a free channel available as a handle. If all the channels are in use, then it returns an error. This happens when the application tries to open more than the specified number of channels. Once the channel creation succeeds, the driver marks that channel to be in use. The driver also fills the additional status field supplied before returning.

2.2.3 Channel Deletion

This function will be used by the class driver to close an open channel. This takes a handle that was created by the create call. Before returning, the driver marks the channel as free.

2.2.4 Read/Write Operations

The MMC mini-driver itself handles all the read/write operations asynchronously. The read/write requests come to the driver through submit call from the GIO or other class-driver or adapter. All the requests to the driver are queued and handled one at a time. Once the operation is complete, the request is removed from the queue and the callback function is invoked indicating request completion.

The application can read/write a maximum of one sector data at a time. A read request takes the sector number, the offset and the number of words to read. The data is read from the MMC media and is copied into the application-supplied buffer. At the completion of the operation, the data requested by the application will be present in the buffer and the callback function is invoked.

A write request takes the sector number, the offset and the number of words to write. The data to be written to the MMC media will be present in the application-supplied buffer. The data in the buffer will be written to the MMC media and after the completion of this operation, callback function is invoked.

2.2.5 Flush/Abort Operations

The MMC driver allows the application to flush or abort the pending request on a channel. The current operation is continued until completion. The pending read/write requests are flushed/aborted (depending on the operation requested). The driver traverses through the request queue and identifies the requests for this channel.

If the operation is flush, all pending write requests are completed and all pending read requests return with IOM_FLUSHED.

If the operation is abort, all the pending requests (input and output) are aborted and return with a status of IOM_ABORTED.

In both cases, the current operation is allowed to complete normally.

2.2.6 Control Commands

This driver supports two control commands. MEDIA_CTRL_GETMEDIATYPE is for querying the media type supported by this driver. The other is to provide the ability to re-initialize the MMC controller peripheral called MEDIA_CTRL_REINITIALIZE.

In case an irrecoverable error is reported to the application, it must invoke MEDIA_CTRL_REINITIALIZE before any further interaction with the driver. Invoking this resets the device.

2.3 Packet Processing

The mini-driver submit function is invoked to perform any (read/write/abort/flush) operation. For example, the GIO class driver passes an IOM packet for each operation. The packet carries the information required to carry out the operation including the command, data, size and status.

The MMC mini-driver looks at the command field and takes necessary action. All the requests are processed asynchronously. If it is a read/write operation, a request is added to the end of the queue before returning. The IOM packets are processed in FIFO order and when the operation is complete, the status field is updated with the status of completion and the callback function is invoked indicating to the class driver that this packet can be freed (or used again). The status will be filled with one of the following values:

- IOM_COMPLETED
- IOM_PENDING
- IOM_ABORTED
- IOM_FLUSHED

For an abort command, the driver completes all the pending read/write requests by invoking the callback function with a status of IOM_ABORTED and in the case of the GIO class driver frees these packets (i.e., places on a free queue for reuse).

For a flush command, the driver completes all the pending read requests by invoking the callback function with a status of IOM_FLUSHED and the GIO class driver frees these packets. All the write requests are completed.

In case of abort and flush, the current operation is continued without stopping.

2.4 Queue Management

The MMC driver handles all read/write requests asynchronously. To achieve this, it maintains a queue of the requests that contains the requests for all the channels opened. Requests get processed in the order of their arrival (FIFO). New requests are added at the tail and requests are removed from the head of the queue for processing. The request queue holds the list of pending requests.

Initially, as shown in Figure 2 (a), the request queue is empty.

Assume there are two channels in operation – channel 1 and channel 2, both requesting for some read/write operations. The queue would look like Figure 2 (b).

If an application requests a FLUSH operation on a particular channel, then the queue is traversed and all the read requests are removed. For example, if the application calls flush on channel1, the queue will look like Figure 2 (c).

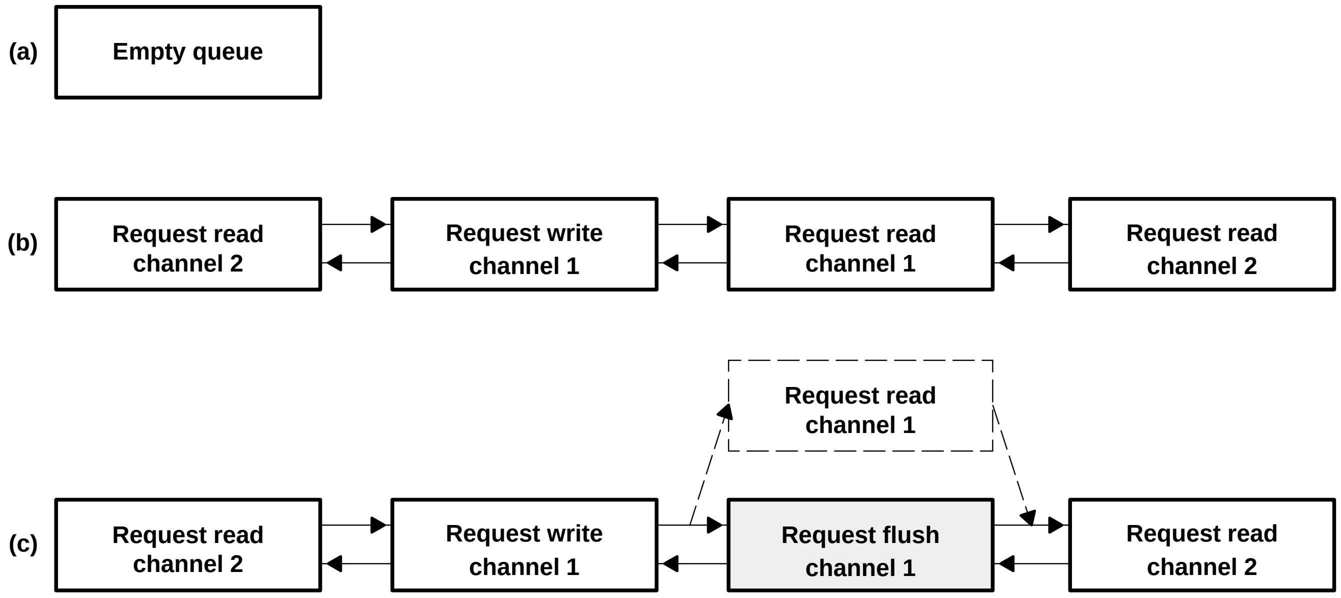

Figure 2. Request queue after flush operation

The driver waits until the pending write operation on channel 1 is completed before it returns the flush call.

If an application asks for an ABORT operation on a particular channel, all the read/write requests are removed from the queue.

For the flush and abort operations, the operation that is in progress will not be stopped. Flush is called by the GIO class driver before deleting when the application requests a channel to be closed.

The DSP/BIOS Que_Obj is used to handle the queue management.

2.5 Interrupt Handling

The MMC controller on C5509 generates an interrupt for the conditions listed in Table 2. The MMC driver handles all these interrupts.

Table 2. Interrupts

Interrupt	Description
DRRDY	Data receive ready interrupt
DXRDY	Data transmit ready interrupt
CRCRS	Response CRC errors interrupt
CRCRD	Read-data CRC errors interrupt
CRCWR	Write-data CRC error interrupt
TOUTRS	Response time-out interrupt
TOUTRD	Read-data time-out interrupt
RSPDNE	Response/command done interrupt
BSYDNE	Busy done interrupt
DATDNE	Data transfer done interrupt

3 Constraints

- This device driver currently operates only in non-DMA mode, does not operate in DMA mode.
- This driver does not support multiple MMC cards in the system.

4 References

1. *DSP/BIOS Driver Developer's Guide* (SPRU616)
2. *TMS320C55x DSP Peripherals Reference Guide* (SPRU317)
3. *TMS320C55x Chip Support Library API Reference Guide* (SPRU433B)
4. *TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU404D)

Appendix A Device Driver Data Sheet

A.1 Device Driver Library Name

- C5509_mmc.l55 (small memory model)
- C5509_mmc.l55l (large memory model)

A.2 DSP/BIOS Modules Used

- HWI – Hardware Interrupt Manager
- QUE – Queue Manager

A.3 DSP/BIOS Objects Used

QUE_Obj

A.4 CSL Modules Used

MMC module – for MMC controller

A.5 CPU Interrupts Used

MMC controller interrupt (HWI interrupt vector #7 for devId == 0x0 and HWI interrupt vector #13 for devId == 0x1) on C5509.

A.6 Peripherals Used

On-chip MMC controller

A.7 Interrupt Disable Time

Maximum time that hardware interrupts can be disabled by the driver: 92 cycles. This measurement is taken using the compiler option –O3.

A.8 Memory Usage

Table A–1. Device Driver Memory Usage

	Uninitialized memory	**Initialized memory**
CODE	—	2694 (8-bit bytes)
DATA	24 (8-bit bytes)	30 (8-bit bytes)

NOTE: This data was gathered using the sectti command utility.
Uninitialized data: .bss
Initialized data: .cinit + .const
Initialized code: .text + .text:init