A DSP/BIOS PCI Device Driver for TMS320C64xx DSPs

Software Development Systems

ABSTRACT

This application report describes the implementation of a DSP/BIOS™ device driver for the TMS320C64xx DSP’s PCI peripheral. This driver was written in conformance to the DSP/BIOS IOM device driver model and APIs. More information about IOM can be found in documents listed in section 4.

The features of this device driver are:

- it allows applications to flexibly bind channels to either a high priority or low priority queue, and
- it is implemented as a re-entrant driver so that it can simultaneously serve multiple PCI devices.

Contents

1 Driver Usage ................................................................. 2
  1.1 Configuration ......................................................... 3
  1.2 Device Parameters .................................................. 3
  1.3 Channel Parameters ............................................... 4
  1.4 Control Commands .................................................. 4
  1.5 Error Interrupt Processing ........................................ 6

2 Architecture ............................................................... 6
  2.1 Data Structures ....................................................... 6
    2.1.1 PCI Request Block ............................................ 6
    2.1.2 Queue Objects ................................................. 7
    2.1.3 Device Object ................................................ 7
    2.1.4 Channel Object .............................................. 8
  2.2 Events ............................................................... 8
    2.2.1 Packet Submissions ........................................... 8
    2.2.2 Interrupt Processing ....................................... 11

3 Driver Design Constraints .............................................. 15

4 References ............................................................... 15

Appendix A Device Driver Data Sheet .................................. 16
  A.1 DSP/BIOS Modules Used ........................................... 16
  A.2 DSP/BIOS Objects To Be Configured ............................. 16
  A.3 CSL Modules Used ............................................... 16
  A.4 CPU Interrupts Used ............................................. 16
  A.5 Peripherals Used .................................................. 16
  A.6 Interrupt Disable Time .......................................... 16
  A.7 Memory Usage ..................................................... 16

Trademarks are the property of their respective owners.
List of Figures

Figure 1    DSP/BIOS Device Driver Model .................................................. 2
Figure 2    I/O Packet Data Structure Extension ........................................... 7
Figure 3    IOM_FLUSH/ABORT Channel X to Queue A .................................... 9
Figure 4    Process Flush and Process Abort in Queue A ............................. 10
Figure 5    IOM_READ/WRITE Submit ............................................................. 11
Figure 6    IOM_READ/WRITE ISR ................................................................. 12
Figure 7    Process Queue .......................................................................... 13
Figure 8    IOM_FLUSH/ABORT ISR ............................................................... 14

List of Tables

Table 1    Device Driver Control Commands ................................................. 5
Table A–1  Device Driver Memory Usage ..................................................... 16

1  Driver Usage

The device driver described here is part of an IOM mini-driver, i.e., it is implemented as the lower layer of a two-layer device driver model. The upper layer is called the class driver and can be either the DSP/BIOS GIO, SIO, or PIP module. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests. Figure 1 shows the overall DPS/BIOS device driver architecture. For more information about the IOM device driver model as well as the GIO, SIO, and PIP modules, see the DSP/BIOS Device Driver Developer’s Guide (SPRU616).
This device driver is a mini-driver that performs master transfers of data between the DSP and other devices or memory on the PCI bus.

This implementation is an example of a “port driver,” in which the driver will not provide a random access window space mapping on behalf of a client device, but rather acts as a port to the PCI bus that provides PCI bus mastering capability. This PCI port driver will service the C64x PCI peripheral on behalf of other device drivers needing PCI access to their device(s) that are connected to the PCI bus.

This in analogous to a DMA device where the client driver submits data transfer requests (i.e., source address, destination address and data size) to the DMA driver, and the DMA driver provides asynchronous notification and status back to the requester via a completion interrupt.

1.1 Configuration

To add this device driver to the DSP/BIOS Configuration Tool, open the Configuration Tool, right-click on the User-Defined Devices icon under the Device Drivers section and select Insert UDEV. From the Objects menu (or by right-clicking on the object), rename the object from UDEV to a unique name for the device driver. Open the Properties dialog for the device you created by right-clicking on the object and modify its properties as follows.

- Init function: C64XX_PCI_init
- Function Table Ptr: C64XX_PCI_FXNS
- Function Table Type: IOM_Fxns
- Device ID: N/A; not used by this driver
- Device params ptr: An optional pointer to an object of type C64XX_PCI_DevParams as defined in the header file c64xx_pci.h. If not specified, the driver will use a default device callback of NULL. Default values are interrupt ID = 4, Error handler = NULL.
- Device global data ptr: N/A; not used by this driver

1.2 Device Parameters

The driver parameters structure allows you to specify, in the DSP/BIOS Configuration Tool, the driver specific parameters when the driver is initialized. The device driver’s parameters are defined as follows:

```c
typedef struct C64XX_PCI_DevParams
{
    Int versionId; /* Version number, set by the app */
    Int pciIntrId; /* interrupt id for normal pci interrupt */
    Int pciErrIntrId; /* interrupt id for pci parity/system error interrupt */
    C64XX_PCI_ErrCallback *errCallback;
    Uns intrMask;
} C64XX_PCI_DevParams;
```

- **versionId**: Version number of the driver.
- **pciIntrId**: The ID for the standard PCI peripheral’s interrupt that lets you specify an interrupt ID different than the default.
• **pciErrIntrId**: The ID for an error interrupt (if implemented).
• **errCallback**: A pointer to a callback function that will be executed for the C64XX_PCI_TEST_ERROR_HANDLER control function. There is also a parameters structure that allows you to specify channel-specific parameters when the channel is created.
• **intrMask**: Interrupt mask, set in the ISR.

### 1.3 Channel Parameters

The user can pass a pointer to a channel parameters structure through the optArgs argument to the GIO_create call. This device driver’s channel parameter structure is defined as follows:

```c
typedef struct C64XX_PCI_Attrs
{
    Uns queuePriority;
} C64XX_PCI_Attrs
```

• **queuePriority**: This structure member specifies whether the channel is bound to the low priority queue or the high priority queue. The default queue priority is low.

### 1.4 Control Commands

The mini-driver interface function, mdControlChan, is called by the GIO class driver’s GIO_control function to:

• program the PCI peripheral’s EEPROM device,
• set or clear the DSP-to-PCI host interrupt request bit,
• reset the device driver channel, and
• reset the peripheral device.

Table 1 contains all of the control commands that are implemented by the mini-driver. Most of these are passed through calls to the chip support library. Return values are passed back from the mdControlChan call in the in-out parameter eeData, which is a member of the C64XX_PCI_EEPROMData data structure that is passed in the arg argument.
Table 1. Device Driver Control Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arg</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C64XX_PCI_EEPROM_ERASE</td>
<td>EEPROM address to operate on</td>
<td>Erases the 16-bit data at the specified address</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_ERASE_ALL</td>
<td>Ignored</td>
<td>Erases the entire EEPROM</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_IS_AUTO_CFG</td>
<td>Ignored</td>
<td>Tests if the PCI will read configuration values from the EEPROM. Places the value of the PCI peripheral’s EEAI field of the EECTL register in the location pointed to by arg.</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_READ</td>
<td>EEPROM address to operate on</td>
<td>Reads the 16-bit data at the specified address</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_SIZE</td>
<td>Ignored</td>
<td>Returns the code associated with the size of the EEPROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0: No EEPROM present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1: 1K_EEPROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x2: 2K_EEPROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x3: 4K_EEPROM†</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x4: 16K_EEPROM</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_TEST</td>
<td>Ignored</td>
<td>Tests if EEPROM is present</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_WRITE</td>
<td>EEPROM address to operate on</td>
<td>Writes the 16-bit data at the specified address</td>
</tr>
<tr>
<td>C64XX_PCI_EEPROM_WRITE_ALL</td>
<td>Data to be written</td>
<td>Writes the 16-bit data in to the entire EEPROM</td>
</tr>
<tr>
<td>C64XX_PCI_DSP_INT_REQ_SET</td>
<td>Ignored</td>
<td>Sets the DSP-to-PCI interrupt request bit of the PCI peripheral’s RSTSRC register</td>
</tr>
<tr>
<td>C64XX_PCI_DSP_INT_REQ_CLEAR</td>
<td>Ignored</td>
<td>Clears the DSP-to-PCI interrupt request bit of the PCI peripheral’s RSTSRC register</td>
</tr>
<tr>
<td>C64XX_PCI_RESET_CHANNEL</td>
<td>Ignored</td>
<td>Resets the specified channel by causing the callback function of the current PCI job and removing any pending packets</td>
</tr>
<tr>
<td>C64XX_PCI_RESET_ALL_CHANNELS</td>
<td>Ignored</td>
<td>Does the same as above, but for all opened channels</td>
</tr>
<tr>
<td>C64XX_PCI_TEST_ERROR_HANDLER</td>
<td>Error type</td>
<td>Fills in the error status block and calls the bus error handler callback function</td>
</tr>
<tr>
<td></td>
<td>C64XX_PCI_EVT_SYSTEM_ERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C64XX_PCI_EVT_PARITY_ERR</td>
<td></td>
</tr>
</tbody>
</table>

† 6415/6416 devices support the 4K_EEPROM only.
1.5 Error Interrupt Processing

This driver has code that implements processing of special PCI error interrupts, such as those raised by bus error and parity error conditions. The error interrupt processing code will not be compiled into the driver unless the preprocessor variable, PCI_ERROR_ISR_IMPL, is defined.

A data structure specific to error interrupt processing is defined in the driver header file, c64xx_pci.h, and is implemented as follows:

```c
typedef struct C64XX_PCI_ErrInfo {
    Ptr    statusReg;  /* pointer of the status register */
    Ptr    inprogressDstAddr; /* dst addr caused err */
    Ptr    inprogressSrcAddr; /* src addr caused err */
} C64XX_PCI_ErrInfo
```

- **statusReg**: A pointer to the PCI peripheral's status register
- **inprogressDstAddr**: The destination address of the packet that caused the error while the DSP was mastering the PCI bus.
- **inprogressSrcAddr**: The source address of the packet that caused the error while the DSP was mastering the PCI bus.

If a bus error interrupt is asserted, this code will examine the source of the interrupt and then fill in information about the packet currently being processed into the driver’s object structure. The bus error callback is then called so that this packet information can be exported back to the application and acted on in an implementation-specific manner.

2 Architecture

This section describes the overall implementation and design of the C6416 PCI device driver. You should also refer to the documentation in the References section for more details on the DSP/BIOS real time operating system, and the PCI Chip Support Library (CSL).

2.1 Data Structures

2.1.1 PCI Request Block

Class drivers communicate with mini-drivers through the I/O packet (IOP), which describes the request to be carried out and supplies the data necessary to do so. The addr field of the I/O packet can either be used to hold the address of the buffer (that holds the data to be written out or read in), or it can be used to point to a structure that holds driver-specific information and serves as an extension to the I/O packet.

The PCI driver has been implemented to use a driver-specific data structure extension to the I/O packet and is shown in Figure 2.
2.1.2 Queue Objects

The PCI driver maintains two queues that hold lists for packets that have been submitted to the driver for reading or writing: one for high priority submissions and one for low priority submissions. A channel is bound to one of the two queues at creation time. It will be attached to the low priority queue by default, if no value is passed in the channels parameters structure – that is, a value of one of the following:

- C64XX_PCI_QUEUE_PRIORITY_HIGH
- C64XX_PCI_QUEUE_PRIORITY_LOW

By implementing this device driver with two queues instead of one, more flexibility is afforded to applications that have more than one device on the PC bus, or to applications that need to prioritize certain channels over others.

2.1.3 Device Object

The device driver’s device object is a single instance object that maintains the overall state of the driver’s variables. The device object is shown below:

```c
typedef struct PCIDeviceObj
{
    IOM_Packet                  *curPacket;
    QUE_Obj                     highPrioQue;
    QUE_Obj                     lowPrioQue;
    C64XX_PCI_TerrCallback     errCallback;
    Uns                         evtMask;
    C64XX_PCI_ErrInfo          *errInfo;
    Int                         openCount;
} PCIDeviceObj, PCIDeviceHandle;
```

- **curPacket**: Pointer to the I/O packet that is currently in process – that is, a packet that has been submitted for I/O (e.g., reading, writing, flush or abort) but has not yet been completed.
- **highPrioQue/lowPrioQue**: Pointers to the two queue objects that hold the submitted I/O packets.
- **errCallback/errInfo**: Used as an optional extension to the PCI driver for boards that generate interrupts when bus error and parity error occur. These object members will store the callback function that runs and will hold information so that the application can get access to more detailed information on why the error occurred.
- **evtMask**: Event mask for error codes asserted by the error interrupt handler.
- **openCount**: Stores the total number of channels that have been opened in the driver.

### 2.1.4 Channel Object

The device driver’s channel object is used to hold and maintain the state of each driver channel. Therefore, a separate object will be created by the mini-driver’s `mdCreateChan` function for each channel. The channel object is shown below

```c
typedef struct ChanObj {
    QUE_Handle          queue;
    Uns                 writeCount;
    IOM_Packet          *flushAbortIop;
    IOM_TiomCallback    callback;
    Ptr                 callbackArg;
} ChanObj, *ChanHandle;
```

- **queue**: used to hold the handle (i.e., the address) of the high or low priority queue. This address is assigned to the channel which holds the I/O packets submitted to the driver.
- **writeCount**: used to hold the count of the number of I/O packets that are pending to be written out.
- **flushAbortIop**: used to hold the I/O packet specified to flush/abort the channel.
- **callback/callbackArg**: points to the callback function and the callback function argument that is called when the PCI peripheral fires an interrupt.

### 2.2 Events

#### 2.2.1 Packet Submissions

The mini-driver interface function, `mdSubmitChan`, will be called by the class driver with a command embedded in the PCI request packet to perform a read, a write, channel flush, or channel abort. Hardware interrupts will then be disabled for the duration of the processing of the packet.

For the case of a request to read or write, the PCI request packet will be put on the queue that was attached to the channel on which the request was made. If there are no other packets in process, then the new request will be processed immediately – otherwise, the request will remain on the queue until the request packets ahead of it are processed. Packets are processed in sequence by simply taking them off the queue one by one, and setting up a PCI transfer with the packet’s source and destination addresses and its byte count.

When packets are submitted to the driver with a command of Flush, all pending input jobs are to be completed with a status of IOM_FLUSHED and all output jobs are to be completed routinely. When packets are submitted to the driver with a command of Abort, all pending calls are completed with a status of IOM_ABORTED.
Figure 3 through Figure 5 illustrate the sequence of operations for Channel Flush/Abort commands.

![Flowchart](image)

**Figure 3. IOM_FLUSH/ABORT Channel X to Queue A**
Figure 4. Process Flush and Process Abort in Queue A
2.2.2 Interrupt Processing

When this driver’s interrupt service routine is called, it first checks to see what type of interrupt was asserted (e.g., either a PCI master abort, a PCI target abort, or most commonly, a PCI transfer complete). The corresponding interrupt bit is in the PCI peripheral’s interrupt source register (ISR).

The ISR then checks to see if a PCI packet request has been in progress. If so, the channel object’s flushAbortTop member is checked to see if a Flush or Abort command has been issued (see discussion in section 2.2.1). If no Flush or Abort has been issued, then the ISR was asserted because the current packet being processed has completed. And so, the next PCI request packet will be taken off of the queue to be processed and the callback function will then be called.

If a channel Flush or Abort has been called, the ISR will perform a callback for the PCI packet request that is in progress, and decrement the writeCount member of the channel structure accordingly.
Figure 6 through Figure 8 illustrate interrupt processing.

Figure 6. IOM_READ/WRITE ISR
A DSP/BIOS PCI Device Driver for TMS320C64xx DSPs

Figure 7. Process Queue
A DSP/BIOS PCI Device Driver for TMS320C64xx DSPs

Figure 8. IOM_FLUSH/ABORT ISR
3 Driver Design Constraints

- This mini-driver can only be used with the IOM class driver. It is not designed to be used with the PIO or DIO class drivers, because this driver is to be used in tandem with one or more other drivers and not directly from the application using the DSP/BIOS SIO or PIP APIs.
- PCI data transfers are limited to 64 Kbytes.
- The PCI host side driver software must configure the PCI bus.

4 References

1. TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide (SPRU403)
2. DSP/BIOS Device Driver Developer’s Guide (SPRU616)
3. TMS320C6000 Peripherals Reference Guide (SPRU190)
4. TMS320C6000 Chip Support Library API User’s Guide (SPRU401)
5. PCI Local Bus Specification, Revision 2.2. PCI Special Interest Group, http://www.pcisig.com
Appendix A  Device Driver Data Sheet

A.1 DSP/BIOS Modules Used
- HWI – Hardware Interrupt Manager
- MEM – Memory Manager
- QUE – Queue Manager
- SYS – System Services Manager

A.2 DSP/BIOS Objects To Be Configured
None

A.3 CSL Modules Used
- PCI Module
- IRQ Module

A.4 CPU Interrupts Used
Default interrupt used is HWI interrupt vector #4

A.5 Peripherals Used
PCI peripheral

A.6 Interrupt Disable Time
Maximum time that hardware interrupts can be disabled by the driver: 333 cycles. This measurement is taken using the compiler option –O3.

A.7 Memory Usage

<table>
<thead>
<tr>
<th></th>
<th>Uninitialized Memory</th>
<th>Initialized Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td></td>
<td>4160 (8-bit bytes)</td>
</tr>
<tr>
<td>DATA</td>
<td>60 (8-bit bytes)</td>
<td>32 (8-bit bytes)</td>
</tr>
</tbody>
</table>

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

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated