PCI2040 EVM User's Guide

Software Guide

Literature Number: SCPU002
November 1999

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 1999, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This manual is provided to assist the developer in programming the PCI2040 in a system which incorporates a DSP-PCI bridge controller.

How to Use This Manual

This document contains the following chapters:

Chapter 1, *Introduction*, provides a functional overview of the EVM software.

Chapter 2, *EVM Debugger*, describes differences between the EVM debugger and the standard C54x debugger.

Chapter 3, *PCI2040 EVM Host Utilities*, describes and shows how to use the PCI2040 EVM host command-line utilities.

Chapter 4, *PCI2040 EVM Host Support Software*, describes the support software for the host that is provided with the EVM board.

Chapter 5, *PCI2040 DSP Support Software*, describes the EVM DSP board support software by providing APIs and example code for the audio codec library.

Chapter 6, *Daughterboard Memory Access Code Examples*, provides code samples for several different types of daughterboard memory access.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

  ```
  0011  0005  0001         .field    1, 2
  0012  0005  0003         .field    3, 4
  0013  0005  0006         .field    6, 3
  0014  0006               .even
  ```

  Here is an example of a system prompt and a command that you might enter:

  ```
  C:  csr -a /user/ti/simuboard/utilities
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect **"***section name***",** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

  LALK *16–bit constant [, shift]*

  The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here's an example of a list:

  { * | *+ | *− }

  This provides three choices: `*`, `*+`, or `*−`.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value*$_1$ *[, ... , value*$_n$*]*

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Related Documentation From Texas Instruments

PCI2040 DSP–PCI Bridge Controller Data Manual, SCPS048

PCI2040 EVM Hardware Guide, SCPU003

PCI2040 Implementation Guide, SCPU004

Contents

1 Introduction 1-1
1.1 Functional Overview 1-2

2 EVM Debugger 2-1
2.1 EVM-Specific Debugger Configuration 2-2
2.1.1 Identification of Boards by Relative Index 2-2
2.1.2 A Dedicated TBC Is Provided on Each EVM Board 2-2
2.2 Code Composer Debugger Configuration 2-3

3 PCI2040 EVM Host Utilities 3-1
3.1 EVM COFF Loader 3-2
3.2 EVM Confidence Test 3-4
3.3 EVM Board Control 3-6
3.4 EVM Board Reset 3-9

4 PCI2040 EVM Host Support Software 4-1
4.1 Host Support Software Components 4-2
4.2 EVM Low-Level Windows Drivers 4-3
4.3 EVM Win32 DLL API 4-4
4.3.1 EVM Win32 DLL API Data Types and Macros 4-4
4.3.2 EVM Win32 DLL API Functions 4-5

5 PCI2040 EVM DSP Support Software 5-1
5.1 DSP Support Software Components 5-2
5.2 Using the DSP Support Software Components 5-2
5.3 Board Driver API 5-3
5.3.1 Board Driver Macros 5-3
5.3.2 Board Driver Data Types 5-3
5.3.3 Board Driver Functions 5-4
5.4 CODEC Library API 5-27
5.4.1 CODEC Library Macros 5-27
5.4.2 CODEC Data Types and Type Definitions 5-27
5.4.3 CODEC Library Functions 5-28
5.4.4 Sample Code 5-40

6 Daughterboard Memory Access Code Examples 6-1
6.1 Sequential 16-Bit Read Accesses 6-2
6.2 Sequential 32-Bit Read Accesses 6-2
6.3 Consecutive 32-Bit Odd/Even Read Accesses 6-3
6.4 Consecutive 32-Bit Write Accesses 6-3

Figures

1–1 EVM Host Software Block Diagram 1-3

Tables

3–1 Application Loader Utility Command Parameters 3-2
3–2 Board Confidence Test Utility Command Parameters 3-4
3–3 Board Control Utility Command Parameters 3-7
3–4 Board Reset Utility Command Parameters 3-9

Chapter 1

Introduction

This chapter provides a functional overview of the EVM software.

The PCI2040 EVM is a low-cost, general-purpose platform for the evaluation of the PCI2040 DSP-to-PCI bridge and the development, analysis and testing of ’C54x digital signal processor (DSP) algorithms and applications. The EVM allows you to develop applications that interface with the PCI2040 that run on both a host PC and the ’C54x DSP to determine if they meet your application requirements. The EVM hardware design information and software application programming interfaces (APIs) also provide a reference design that facilitates your own PCI2040 hardware and software development.

The EVM is bundled with support software that includes a debugger, Windows 95/98 and NT 4.0 device drivers, host PC and DSP software APIs, example applications with source code, and various utility applications. This hardware/software bundle provides an integrated package that allows you to quickly evaluate the PCI2040 and ’C54x DSP devices and develop applications.

Topic **Page**

1.1 Functional Overview 1-2

1.1 Functional Overview

The EVM software consists of host and DSP support software. The host software supplied with the EVM includes the following host utilities and libraries:

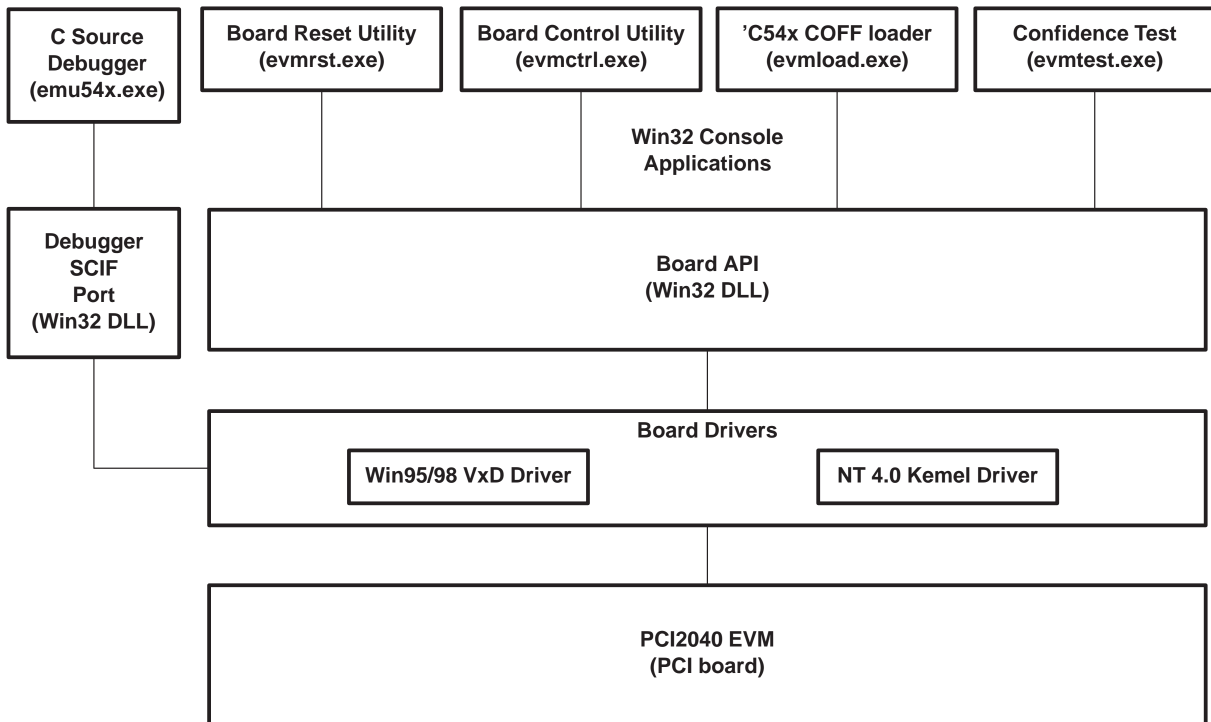
- C Source debugger (emu54x.exe). The debugger software tools help you debug ’C54x software on the board.
- EVM Board reset utility (evmrst.exe). This utility is used to reset the board.
- EVM confidence test utility (evmtest.exe). This utility tests the basic operation of the board.
- EVM board control utility (evmctrl.exe). This utility is used to perform various control and status operations with the board.
- EVM COFF loader utility (evmload.exe). This utility is used to load and execute ’C54x software on the board.
- EVM Win32 DLL (pci2040.dll). The Win32 host libraries consist of a user-mode dynamic linked library (DLL) that supports Windows 95, 98, and Windows NT. This DLL provides user software access for control and communication with the EVM board.
- Example source code. Example code that illustrates how to use the Win32 DLL functions is provided with the PCI2040 EVM.

The host software supplied with an EVM board provides utilities to configure the board, to debug PCI2040 ’C54x software on the board, to load and execute ’C54x software on the board, and to test the basic operation of the board. Also supplied with the EVM board is a Win32 dynamic link library (DLL) that provides user software access for control and communication with the EVM board. The host utilities and host libraries run on an Intel PC under either Windows 95, Windows 98, or Windows NT 4.0. Figure 1–3 provides a block diagram of the EVM host software components and their relationships.

The far left side of the figure shows the components involved in the C source debugger support. The TI C source debugger makes calls to the debugger SCIF component. The SCIF component calls the JTAG TBC API functions to perform emulation functions. The JTAG TBC API calls the low-level (Ring 0) Win 95 and NT drivers to access the JTAG TBC device on the EVM board.

The remainder of the figure shows the Win32 host utilities and the components they use to access the hardware. The Win32 applications call the Windows 95, 98, or Windows NT DLL that implement a consistent Ring 3 API. These DLLs make calls to the Ring 0 drivers which provide access to the hardware.

Figure 1–1. EVM Host Software Block Diagram



The DSP support software supplied with the EVM board includes the following components:

- Board support library (board.c and board.h). This library provides 'C54x board specific routines for EVM configuration and control.
- Codec library (Codec.c and board.h). This library is a collection of routines that configures and controls the operation of the AD77 audio codec device.
- Example source code. Code examples are provided to demonstrate the use of the codec and board support functions.

Chapter 2

EVM Debugger

This chapter describes differences between the EVM debugger and the standard 'C54x debugger.

Topic **Page**

2.1 EVM-Specific Debugger Configuration 2-2

2.2 Code Composer Debugger Configuration 2-3

2.1 EVM-Specific Debugger Configuration

The PCI2040 ’C54x C source debugger differs from the standard (emulator-based) ’C54x debugger in that:

1) It takes advantage of the PCI architecture by accessing the EVM board via standard memory operations, rather than using slower, less flexible I/O port operations. Also, since the memory locations and other board-specific interface parameters are dynamically assigned, it identifies each board merely by a simple relative index, rather than an explicitly configured port address.

2) It accesses the JTAG interface through a dedicated TBC (test bus controller) on each EVM board.

2.1.1 Identification of Boards by Relative Index

The PCI architecture provides an automatic mechanism for uniquely identifying the board type and bus slot location of each installed board. Thus EVM boards are selected simply by using an index that starts at zero for the first EVM board installed and increments up to the total number of EVM boards installed, minus 1. In other words, a single board would be indexed as 0, two boards as 0 and 1, and N boards as 0, 1, 2, … N–1.

Note that the index is not determined by absolute slot location. In particular, note that a single EVM board will always be board 0, no matter which slot it occupies, and that inserting or removing one EVM board from a system may change the index value used to access other EVM boards. For example, consider a system with two EVM boards installed, indexed as 0 and 1. In this system, if board 1 were removed, the other would remain board 0. However, if board 0 were removed, the other board, previously identified as board 1, would then become board 0.

2.1.2 A Dedicated TBC Is Provided on Each EVM Board

The provision of a TBC on each EVM board means that the PCI2040 ’C54x debugger can be used to develop and debug software on one or more PCI2040 EVM boards without using a separate emulator board. (A separate emulator board, i.e., an XDS510, and a standard ’C54x debugger can of course be used with the EVM, but they are not required.) On the other hand, the use of a dedicated TBC for each EVM board means that the DSP’s residing on multiple EVM boards do not share a single test bus, and for this reason they cannot respond synchronously to debugger commands issued to multiple boards in parallel. This means that PDM (parallel debug manager) parallel commands (PHALT, PRUN, PRUNF, PSTEP) are unavailable when using the PCI2040 ’C54x debugger and PDM.

In summary, the PCI2040 ’C54x debugger behaves like a standard ’C54x debugger, except that:

1) When a debugger instance is started, a particular EVM board is selected by its relative board index, rather than an I/O port address.

2) PDM parallel commands (PHALT, PRUN, PRUNF, PSTEP) are unavailable.

2.2 Code Composer Debugger Configuration

The PCI2040 EVM support software includes drivers for both version 3.x and 4.x of TI’s code composer IDE debuggers. These drivers are stored in the \pci2040\Code Composer\3.x and \pci2040\Code Composer\4.x directories, respectively. Copy the appropriate driver (ti2040nt.dll for 3.x or ti2040nt.dvr for 4.x) into the code composer’s bin directory, or into the directory where the other code composer drivers are stored on your system.

Before you can run the code composer, you must run the code composer setup application to select the PCI2040 EVM driver. It is important to select a port I/O address of 0 (not the default of 0x240), or the debugger will not run. If you have multiple PCI2040 EVM boards installed, select port I/O addresses of 1, 2, etc.

Chapter 3

PCI2040 EVM Host Utilities

This chapter describes and shows you how to use the following PCI2040 EVM host command-line utilities:

- EVM COFF loader utility
- EVM confidence test utility
- EVM configuration utility
- EVM reset utility

Topic **Page**

3.1 EVM COFF Loader 3-2

3.2 EVM Confidence Test 3-4

3.3 EVM Board Control 3-6

3.4 EVM Board Reset 3-9

3.1 EVM COFF Loader

This host-based command line utility provides a way to load and execute a DSP application without requiring the user to write a custom utility or use the debugger. It is accomplished using the PCI2040's PCI bus interface between the host and the target DSP. The utility uses the board's Win32 user-mode DLL to interface with the board.

The loader utility also provides the host side of the application's file I/O once execution begins.

The source code for this utility serves as a good example on how to use the Win32 DLL to load and execute a DSP application.

The utility is invoked with the following syntax:

evmload *filename* [*options*]

evmload Command that invokes the DSP application loader utility

filename Name of DSP COFF application file to load and execute

options Options that affect the way the utility behaves. Options are not case-
sensitive and can appear anywhere on the command line following the command.

If no options are entered, the utility displays the help screen to identify the available command parameters.

Table 3–1 summarizes the DSP application loader utility command parameters. Command parameters can be used in any order on the command line.

Table 3–1. Application Loader Utility Command Parameters

Command Parameter	Description
–? or –h	Utility help display
–b *num*	Selects specific EVM target board; *num* is zero-based relative board index, ranging from 0 (first board) to n–1 (last board), where n boards are installed. Default is 0.
–c	Clears the .bss section to 0.
–d	Displays all COFF section data in ASCII via the standard output stream which can be a large amount of data. The –d option enables the –v (verbose) option.
–dr	Assume DROM is asserted (SARAM2 is data).
–i *filename*	Filename of the host file to be read and sent to the DSP as file input data. Host to DSP stream is not opened when omitted. The default filename for this option is input.dat.
–ls *bytes*	Size limit of output file. Saving data to the output file using the –o option will terminate when this number of bytes is reached. The default is 0 (no size limit).
–lt *seconds*	Time limit of output file. Saving data to the output file using the –o option will terminate when this number of seconds is reached. The default is 0 (no time limit).

Table 3–1. Application Loader Utility Command Parameters (Continued)

–o *filename*	Filename of the host file to be written with data received from the DSP as file output data. DSP to host stream is not opened when omitted. The default filename for this option is output.dat.
–q	Suppresses output to the display (quiet)
–s	Shows only; writes data to standard output stream but does not write data to DSP memory. This option enables the –v option.
–v	Enables verbose mode, in which basic COFF section information is displayed via the standard output stream.
–n *dspNum*	Selects target DSP 0–3. Defaults to 0.
–z	Displays loader status and waits for key to be pressed after each step.

The loader generally performs the following steps:

1) If the *Show* option is selected, the loader does not actually load an image, but just reads the COFF file and sends output to *stdout* only, and then stops. The information displayed depends on whether the –d (dump) option is selected:

- ❑ If the –d option is not selected, the information sent to *stdout* consists of a descriptive line for each section, including name, size, and flags.
- ❑ If the *Dump* option is selected, all of the data in each section will be displayed, which may be a very large amount of data and, as such, may take a long while to display.

If the –S option is not specified, the loader continues to step 2.

2) Opens a driver connection to a specific EVM target board.

3) Performs a board reset.

4) Performs a DSP reset.

5) Opens an HPI connection.

6) Uses the HPI to load an executable image from a 'C54x COFF file, clearing the bss section (if the –c option is specified).

7) Releases the DSP from the halted reset state, thus starting program execution.

8) Closes the driver connection to the EVM target board.

An example load file, blink.out, is provided for rudimentary demonstration of loader operation. To load and execute it, simply enter the following command line:

evmload blink.out

3.2 EVM Confidence Test

This confidence test is a command line utility that provides a way to test the PCI2040 EVM board, enabling the user to verify proper installation and operation of the board. The testing includes checkout of the PCI2040, DSP, external memory and audio codec. This automated utility provides pass/fail indications for each of these items. This utility provides the user with confidence that the board is working properly.

This test must be executed from the directory into which it was installed since several support files are required for the test operations.

The board confidence test utility is invoked with the following syntax:

evmtest [options]

evmtest Command that invokes the board confidence test utility

options Options that affect the way the utility behaves. Options are not case-sensitive and can appear anywhere on the command line following the command.

If no options are entered, the utility continues to run.

Table 3–2 summarizes the DSP application loader utility command parameters. Command parameters can be used in any order on the command line.

Table 3–2. Board Confidence Test Utility Command Parameters

Command Parameter	Description
Log_filename	Name of optinal file for test results logging
–? or –h	Utility help display
–b *num*	Selects the specific EVM target board; *num* is zero-based relative board index, ranging from 0 (first board) to n–1 (last board), where n boards are installed. Default is 0.
–d	Selects specific DSP on the board.
–i	Enables information-only mode. In this mode, only the board configuration information is displayed, bypassing the board tests.

The confidence test checks for proper installation of the Win32 DLL and low-level driver by opening a connection to the board. With successful access to the board, the boards configuration information is retrieved and displayed to the command window. If the information only mode is requested (–i option), the program terminates at this time.

The configuration information includes the:

- Board Index
- Board type and revision

The tests performed on the board include:

- Memory tests
- ’C54x interrupts tests
- Audio codec
  - Left channel tone test
- Right channel tone test
  - Left/right channel tone test
  - Line in loopback test
  - Mic in loopback test
- CPLD semaphores
- PCI controller and board LEDs
- JTAG

Upon completion of the tests, the board is reset. All of the configuration and test information is written to stdout and optionally recorded to a file specified on the command line.

3.3 EVM Board Control

This host-based command line utility provides a way to configure, control, and monitor the PCI2040 EVM board without requiring the user to write a custom utility. This is accomplished without the use of an emulator or the JTAG port. It is accomplished using the PCI2040's PCI bus interface between the host and the target DSP. The utility uses the board's Win32 user-mode DLL to interface with the board.

The source code for this utility serves as a good example on how to use the Win32 DLL to interface with the PCI2040 EVM.

The board control utility can:

- ❑ Display board's PCI configuration
- ❑ Display PCI2040 device configuration information
- ❑ Configure the board and PCI2040 device
- ❑ Control the DSP and board resets
- ❑ Display and control hotswap signals
- ❑ Interrupt 'C54x DSP
- ❑ Read and write 'C54x DSP memory

The utility is invoked with the following syntax:

evmctrl [*options*]

evmctrl Command that invokes the board control utility

options Options that affect the way the utility behaves. Options are not case-sensitive and can appear anywhere on the command line following the command.

If no options are entered, the utility displays the help screen to identify the available command parameters.

It is envisioned that a special version of this utility, created with conditional compilation, can be used as the basis of the board verification software.

Table 3–3 summarizes the board control utility command parameters. Command parameters can be used in any order or combination on the command line. Additional low-level access parameters may be created to support board verification, but these would not be activated in the release provided to users.

Table 3–3. Board Control Utility Command Parameters

Command Parameter	Description
–? or –h	Utility help display
–b *num*	Selects specific EVM target board; *num* is zero-based relative board index, ranging from 0 (first board) to n–1 (last board), where n boards are installed. Default is 0.
–bi	Display board information
–cr *offset*	Read PCI2040 device configuration register *offset* is the PCI2040 register offset.
–cw *offset data*	Write PCI2040 device configuration register. *offset* is the PCI2040 register offset. *data* is the data value to write.
–cp	Display CPLD registers.
–hp	Display PCI2040 HPI configuration
–hs [*0 | 1]*	Display hot swap status led parameter can be used to turn the hotswap LED ON (1) or OFF (0).
–id *intr*	Interrupt DSP *intr* selects type of interrupt HPI DSPINT – 0 NMI – 1 DSPINT3 – 2
–l *filename*	Load DSP application *filename* is the name of the COFF file to be processed and loaded. Note: This provides a basic COFF load operation without the user control provided by the evmload utility.
–mr *addr type [count]*	Read and display DSP memory at address indicated by *addr*. *type* specifies PROGRAM (0) or DATA (1) memory. If the optional *count* value is used, a block of DSP memory can be read and displayed. The values of *addr* and *count* can be decimal or hexadecimal (0x).
–mw *addr type data* [*count*]	Write or fill DSP memory at address indicated by *addr*. *type* specifies PROGRAM (0) or DATA (1) memory. The data value to write is *data*, and the number of locations to write (fill) is indicated by *count*.

Table 3–3. Board Control Utility Command Parameters (Continued)

Command Parameter	Description
–p	Displays board PCI configuration information
–rb	Resets the board
–q	Suppresses the banner and progress information
–n *dspNum*	Selects target DSP 0–3. Defaults to 0.
–rt *[0 / 1]*	Reset JTAG TBC. (0=unreset, 1=reset). Default is reset.
–rd [*0* / *1*]	Controls the DSP's reset. If *reset* is 1, the DSP is held in reset. If *reset* is 0, the DSP is released from reset. If *reset* is not used, a reset pulse is generated.

3.4 EVM Board Reset

This host-based command line utility provides a way to reset the board and restore it into a known state. This is accomplished without the use of an emulator or the JTAG port. It is accomplished using the PCI2040's PCI bus interface between the host and the target DSP. The utility uses the board's Win32 user-mode DLL to interface with the board.

The source code for this utility would serve as a good example on how to use the Win32 DLL to interface with the PCI2040 EVM.

The utility is invoked with the following syntax:

evmrst [*options*]

evmrst Command that invokes the board control utility

options Options that affect the way the utility behaves. Options are not case-sensitive and can appear anywhere on the command line following the command.

If no options are entered, the utility displays the help screen to identify the available command parameters.

It is envisioned that a special version of this utility, created with conditional compilation, can be used as the basis of the board verification software.

Table 3–4 summarizes the board control utility command parameters. Command parameters can be used in any order or combination on the command line. Additional low-level access parameters may be created to support board verification, but these would not be activated in the release provided to users.

Table 3–4. Board Reset Utility Command Parameters

Command Parameter	Description
–? or –h	Utility help display
–b *num*	Selects specific EVM target board; *num* is zero-based relative board index, ranging from 0 (first board) to n–1 (last board), where n boards are installed. Default is 0.
–n dspNum	Selects specific DSP on the board
–q	Quiet mode

Chapter 4

PCI2040 EVM Host Support Software

This chapter describes the support software for the host that is provided with the EVM board. This software works on Pentium-based PCs running either Windows 95, Windows 98, or Windows NT 4.0. The software is installed during the EVM board installation.

With the provided low-level driver and user DLL, a user application on the host can:

- Reset and configure the ’C54x
- Load and execute code
- Send and receive messages as well as data streams
- Access board resources via the HPI
- Interrupt the DSP
- Configure the PCI controller

You can use the DLL functions described in section 4.3.2, *EVM Win32 DLL API Functions*, to perform all these operations.

Topic **Page**

4.1 Host Support Software Components 4-2

4.2 EVM Low-Level Windows Drivers 4-3

4.3 EVM Win32 DLL API 4-4

4.1 Host Support Software Components

The host software components consist of an operating-system specific low-level driver and a user-mode Win32 DLL. The Win32 DLL provides a consistent API for both Windows 95 and Windows NT 4.0 board access through the low-level driver. These components are used to create and execute user mode applications for the EVM board. By using the Win32 DLL, you can write user mode Win32 applications that execute under both operating systems.

These components, along with user mode host example code, are installed during the EVM installation (refer to the *PCI2040 EVM Hardware Guide*, Section 2.6, *EVM Software Installation*, for more information).

4.2 EVM Low-Level Windows Drivers

A low-level driver that is specific to the supported operating system handles all direct access to the one or more EVM boards in a system. For Windows 95 and 98, windows VxD, pci2040.vxd,is the low-level driver that provides access to the EVM hardware. For Windows NT 4.0, a kernel mode driver, pci2040.sys, provides the EVM hardware access.

All the functionality required for control and communication with the EVM hardware is provided by a Win32 DLL, pci2040.dll, which handles the details of the low-level driver access. This DLL presents a common Win32 API for Windows 95, Windows 98, and Windows NT 4.0 user applications. Thus a Win32 user mode application using the Win32 DLL executes under any operating system.

4.3 EVM Win32 DLL API

This section describes the EVM Win32 DLL data types and provides summaries of the Win32 DLL API functions.

4.3.1 EVM Win32 DLL API Data Types and Macros

(a) Board Types

```
typedef enum {
   TYPE_UNKNOWN = 0,
   TYPE_PCI2040_C54XEVM,
} PCI2040DLL_BOARD_TYPE, *PPCI2040DLL_BOARD_TYPE;
```

PCI2040DLL_BOARD_TYPE type definition defines the possible values returned by the pci2040_board_type() function. A properly functioning EVM board will return a board type of TYPE_EVM.

(b) DSP-to-Host Events

```
typedef enum {
   DSP0_INT = 0,
   DSP1_INT,
   DSP2_INT,
   DSP3_INT,
   GP_INT,
   GPIO2_INT,
   GPIO3_INT,
   HPIERROR_INT,
   HPIERROR0_INT,
   HPIERROR1_INT,
   HPIERROR2_INT,
   HPIERROR3_INT
} PCI2040_HOST_EVENT, PPCI2040_HOST_EVENT;
```

PCI2040_HOST_EVENT type definition defines the possible interrupts that can be sent from the board to the host. These enumerations are used in the pci2040_get_event_handle() call to obtain the corresponding event handle.

(c) Host-to-DSP Events

```
typedef enum {
   NMI_INT = 0,
   DSP_INT,
   DSP_INT3
} PCI2040_DSP_EVENT, PCI2040_DSP_EVENT;
```

PCI2040_DSP_EVENT type definition defines the possible interrupts that can be sent to the board by the host. These enumerations are used in the pci2040_generate_int () call to interrupt the DSP.

(d) CPLD Semaphores

```
typedef enum {
   PCI2040_SEM0,
   PCI2040_SEM1
} PCI2040_SEM, PPCI2040_SEM;
```

PCI2040_SEM type definitions defines the CPLD semaphores. These enumerations are used in the pci2040_sem_wait() call to acquire the corresponding user semaphore.

(e) Device Base Address

```
typedef enum {
   CSR_BAR = 0,
   HPI_BAR,
   TBC_BAR,
   CPLD_BAR
} PCI2040_BAR, *PPCI2040_BAR;
```

PCI2040_BAR type definition enumerates the different base addresses used by the device driver. These enumerations are used in the pci2040_get_base_address() call to get the corresponding base address.

(f) Macros

```
#define   PROG_MEMORY        0
#define   DATA_MEMORY        1
#define   IO_MEMORY          2
#define   NO_FIFO_TIMEOUT    0
```

4.3.2 EVM Win32 DLL API Functions

Function	Description
pci2040_reset_dsp()	Reset the DSP on a board and set boot mode
pci2040_unreset_dsp()	Release the DSP from it’s reset state
pci2040_fifo_read()	Read data from a board’s fifo interface
pci2040_fifo_write()	Write data to a board’s fifo interface
pci2040_coff_display()	Display COFF section information
pci2040_hpi_open()	Open the HPI for a DSP
pci2040_hpi_close()	Close the HPI for a DSP
pci2040_hpi_read()	Read DSP memory using the HPI
pci2040_hpi_write()	Write DSP memory using the HPI
pci2040_hpi_read_single()	Read a single 16-bit value from DSP memory
pci2040_hpi_write_single()	Write a single 16-bit value to DSP memory
pci2040_hpi_fill()	Fill DSP memory using the HPI
pci2040_hpi_generate_int()	Interrupt a DSP using the HPI

Function	Description
pci2040_set_timeout()	Set the fifo data transfer time out value
pci2040_init()	Initialize the PCI controller (PCI2040) registers
pci2040_get_configuration()	Retrieves the on-board PCI controller information
pci2040_store_configuration()	Writes the on-board PCI controller information
pci2040_enable_int()	Enables a specific interrupt
pci2040_disable_int()	Disables a specific interrupt
pci2040_hotswap_switch()	Returns the status of the hot swap switch
pci2040_hotswap_led_on()	Turns on the hot swap LED
pci2040_hotswap_led_off()	Turns off the hot swap LED
pci2040_printf_control()	Sets the printf redirection function
pci2040_input_control()	Sets the input redirection function
pci2040_dll_revision()	Returns the DLL revision
pci2040_read_reg()	Reads a PCI controller register
pci2040_write_reg()	Writes a PCI controller register
pci2040_dsp_present()	Determines if a specific DSP is present
pci2040_mem_config()	Sets the DSP memory configuration
pci2040_cpld_read_all()	Dumps the CPLD state
pci2040_get_event_handle()	Returns the handle for a specific event
pci2040_read_cpld_reg()	Reads a CPLD register
pci2040_write_cpld_reg()	Writes a CPLD register
pci2040_sem_wait()	Waits indefinitely for a semaphore
pci2040_user_semaphore_get()	Acquires a semaphore
pci2040_user_semaphore_release()	Releases a semaphore
pci2040_get_base_address()	Returns a specific base address
pci2040_tbc_reset()	Reset the test bus controller

pci2040_open — Open a driver connection to a board

Syntax

<pci2040dll.h.h>

HANDLE **pci2040_open**(int board_index, BOOL exclusive_flag);

Description

The pci2040_open function opens a driver connection to a specific EVM. The returned HANDLE is used for all further accesses to the target board.

- The *board_index* parameter is a zero based relative index. Valid index values depend on the number of EVM and McEVM boards in a system and will range from 0 to n–1, where n is the number of boards.
- The *exclusive_flag* parameter indicates an exclusive open request of the target board. An exclusive open will fail if the target board is currently open and additional open requests will fail for a target board that has been opened exclusively.

Return Value

The function returns a HANDLE that is to be used for all further accesses to the target board. A return value of INVALID_HANDLE_VALUE indicates a failure.

Example

In the following example, the pci2040_open function opens an exclusive connection to board 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
. . .
   pci2040_close(hBoard);
```

pci2040_close — Close a driver connection to a board

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_close**(HANDLE hDevice);

Description

The pci2040_close function closes a previously opened driver connection to a board. The returned value is TRUE for a successful operation.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_close function closes a previously opened driver connection to an EVM board.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
. . .
   pci2040_close(hBoard);
```

pci2040_board_type — Retrieve board type and version information

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_board_type**(HANDLE hDevice, PEVM6X_BOARD_TYPE p_board_type, PULONG p_rev_id);

Description

The pci2040_board_type function gets the board type and revision of a target board as stored in the PCI configuration space of the board. The return value indicates the success of the function call.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *p_board_type* and *p_rev_id* parameters are pointers to the locations to place the requested information.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_board_type() function retrieves information about the board type of the open board.

```
#include <windows.h>
#include <pci2040dll.h>
#include <stdio.h>
...
   int                     board_index;
   HANDLE                  hBoard;
   ULONG                   revId;
   PCI2040DLL_BOARD_TYPE boardType;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
   if ( !pci2040_board_type( hBd, pBdInfo, revId ) )
   {
      printf("Error:pci2040_board_type_failed.\n");
   }
   else
   {
      if (boardType == TYPE_PCI2040_C54XEVM)
      {
         printf("PCI2040 C54x Board,  Revision: %d\n.",
            boardType, revId );
      }
      else
      {
         printf("Unknown board type.\n");
      }
   }
   else
. . .
   pci2040_close(hBoard);
```

pci2040_reset_board — Reset a board

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_reset_board**(HANDLE hDevice);

Description

The pci2040_reset_board function causes a hardware reset on the target board. This function will fail unless the target board has been opened exclusively. The return value indicates the success of the function call.

- The *hDevice* parameter is the handle returned from a successful exclusive pci2040_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_reset_board function resets a board previously opened exclusively.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard= pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
   if (!pci2040_reset_board(hBoard))
   {
      //board reset failed
      exit(-1);
   }
   else
   {
      //board reset succeeded
   }
```

pci2040_reset_dsp — Reset a DSP

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_reset_dsp**(HANDLE hDevice, ULONG dspNum);

Description

The pci2040_reset_dsp function causes a reset to only the DSP specified by dspNum. The DSP will be left in a halted state. To allow the DSP to begin executing, use the pci2040_unreset_dsp function. The return value indicates the success of the function call.

- The *hDevice* parameter is the handle returned from a successful exclusive pci2040_open call.
- The *dspNum* parameter is the DSP to reset.

The usual way to load and start a DSP is:

1) reset the DSP.
2) load the DSP program with HPI write calls. This can be accomplished with the pci2040_coff_load function.
3) unreset the DSP using the pci2040_unreset_dsp function

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the function pci2040_reset_dsp() function holds the DSP in reset. In this state the DSP still accepts HPI accesses. The pci2040_unreset_dsp() function releases the DSP from this state and begins execution of DSP code.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
   if (!pci2040_reset_dsp(hBoard, dspNum))
   {
      //board reset failed
      exit(-1);
   }
```

pci2040_unreset_dsp — Releases DSP from the reset state

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_unreset_dsp**(HANDLE hDevice, ULONG dspNum);

Description

The pci2040_unreset_dsp function releases the DSP from the halted state invoked by the pci2040_reset_dsp function. This function should only be used in conjunction with a pci2040_reset_dsp call. The return value indicates the success of the function call.

- The *hDevice* parameter is the handle returned from a successful exclusive pci2040_open call.
- The *dspNum* parameter is the DSP to reset.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_unreset_dsp function releases the DSP from the halted state.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LONG      dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
   pci2040_reset_board(hBoard);
   pci2040_coff_load(hBoard, NULL, “example.out”, FALSE,
      FALSE, FALSE);
   if (!pci2040_unreset_dsp(hBoard, dspNum))
   {
      //board reset failed
      exit(-1);
   }
```

pci2040_fifo_read — Read data from a board's FIFO interface

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_fifo_read**((LPVOID hpi,
PULONG pbuffer,
PULONG plen,
ULONG dspNum);

Description

The pci2040_fifo_read function transfers data from the DSP to the host using the memory implemented FIFO. This transfer can take an indeterminate amount of time depending on the DSP making data available for the transfer. To prevent the host from pending on a transfer for too long, a time-out feature is available. Use the pci2040_set_timeout function to set the time-out value.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be filled by the read operation. This address must be 16-bit word aligned.
- The *p_length* parameter is the address of the read length. The length will be updated with the actual transfer length. The length is the number of bytes to be transferred, but must be a multiple of 2 since all transfers are 16-bit words.
- The *dspNum* parameter specifies the DSP to read from.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_fifo_read() function reads data from the FIFO.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   LONG      len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
. . .
   if (!pci2040_fifo_read(hHpi, (ULONG)buffer, &len,
      dspNum))
   {
      //FIFO write failed
      exit(-1);
   }
```

pci2040_fifo_write — Write data to a board’s FIFO interface

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_fifo_write**(LPVOID hpi, PULONG pbuffer,
PULONG plen, ULONG dspNum);

Description

The pci2040_fifo_write function transfers data from the host to the DSP using the memory implemented FIFO. This transfer can take an indeterminate amount of time depending on the DSP making data available for the transfer. To prevent the host from pending on a transfer for too long, a time-out feature is available. Use the pci2040_set_timeout function to set the time-out value.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be filled by the read operation. This address must be 16-bit word aligned.
- The *p_length* parameter is the address of the read length. The length is updated with the actual transfer length. The length is the number of bytes to be transferred, but must be a multiple of 2 since all transfers are 16-bit words.
- The *dspNum* parameter specifies the DSP to read from.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_fifo_write function writes data to the FIFO.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   LONG      len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
. . .
   if (!pci2040_fifo_write, (ULONG)buffer, &len, dspNum))
   {
      //FIFO write failed
      exit(-1);
   }
```

pci2040_coff_load — Load a COFF image to a board using the HPI

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_coff_load**(HANDLE hDevice, LPVOID hpi, char *filename,
BOOL verbose_flag, BOOL clear_bss_flag,
BOOL dump_flag, ULONG dspNum);

Description

The pci2040_coff_load function reads a COFF image and writes the data to DSP memory using the HPI. This function allows the user to load data or executable images from a COFF file to DSP memory.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *hpi* parameter is either NULL or the handle returned from a successful pci2040_open_hpi call.
- The *filename* parameter is the filename of the COFF file to be processed.
- The *verbose_flag* parameter if TRUE will cause COFF file information to be sent to standard out during COFF file processing.
- The *clear_bss_flag* parameter if TRUE will cause the bss section to be set to zero. This is not the default behavior of the DSP debugger.
- The *dump_flag* parameter if TRUE will cause the display to standard out of all the data being written to DSP memory. This can be a very large amount of data.
- The *dspNum* parameter specifies the DSP to load the COFF file onto.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_coff_load() function loads a COFF executable to a DSP. The COFF executable is the file example.out, and the verbose flag is set, which causes COFF section information to be displayed to stdout during the load operation.

```
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (hBoard == INVALID_HANDLE_VALUE)
   {
      //board open failed
      exit(-1);
   }
pci2040_reset_dsp(hBoard, dspNum);
if !(pci2040_coff_load(hBoard, NULL, “example.out”, FALSE,
FALSE, FALSE, dspNum))
   {
      //COFF load failed
      exit(-1);
   }
   pci2040_unreset_dsp(hBoard, dspNum);
```

pci2040_coff_display — Display COFF information

Syntax

#include <pci2040dll.h>

BOOL **pci2040_coff_display**(char *filename,
BOOL clear_bss_flag,
BOOL dump_flag);

Description

The pci2040_coff_display function outputs COFF file information to standard out. This information includes details for each section, including name, size, and flags.

- The *filename* parameter is the filename of the COFF file to be processed.
- The *clear_bss_flag* parameter if TRUE will cause the bss section to be set to zero. This is not the default behavior of the DSP debugger.
- The *dump_flag* parameter if TRUE will cause the display to standard out of all the data being written to DSP memory. This can be a very large amount of data.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_coff_display function displays the section information of a COFF file to stdout.

```
#include <windows.h>
#include <pci2040dll.h>.
. . .
   if (!pci2040_coff_display(“example.out”, FALSE,
   DISPLAY))
   {
      //COFF display failed
   }
```

pci2040_hpi_open — Open the HPI for a board

Syntax

#include <pci2040dll.h.h>

LPVOID **pci2040_hpi_open**(HANDLE hDevice);

Description

The pci2040_hpi_open function establishes a single connection per target board to the HPI of a target board. After the HPI has been successfully opened, read and write operations can be performed to DSP memory.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.

Note:

The various HPI accesses performed to a board's HPI are protected by a MUTEX. This prevents multiple operations from interfering with each other. But, this also means that an operation may not begin immediately if another HPI operation is in progress. This could cause a HPI interrupt to the DSP to be delayed.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_open() function opens the HPI on the EVM board. This handle is required for access to DSP memory.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   LONG      len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
   if (!pci2040_hpi_read(hHpi, buffer, &len, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_read() failed
}
```

pci2040_hpi_close — Close the HPI for a board

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hpi_close**(LPVOID hpi);

Description

The pci2040_hpi_close function closes an open HPI session started with a successful pci2040_hpi_open call.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_close() function closes the HPI after reading the DSP memory.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   LONG      len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
   if (!pci2040_hpi_read(hHpi, buffer, &len, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_read() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_hpi_read — Read DSP memory using the HPI

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hpi_read**(LPVOID hpi, PULONG p_buffer,
PULONG p_length, ULONG src_addr,
int memSpace, ULONG dspNum);

Description

The pci2040_hpi_read function transfers data from the target DSP memory space to host memory. The HPI accesses DSP memory from the host.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be filled by the read operation. This address must be 32-bit word aligned.
- The *p_length* parameter is the address of the read length. The length is updated with the actual transfer length. The length is the number of bytes to be transferred, but must be a multiple of 2 since all transfers are 16-bit words.
- The *src_addr* parameter is the transfer starting address in the DSP's memory space. This address is from the DSP's point of view.
- The *memSpace* parameter specifies the either program or data space.
- The *dspNum* parameter specifies the DSP to read from.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_read() function reads 200 16-bit words from DSP 0's memory at address 0xf8.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   LONG      len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
   if (!pci2040_hpi_read(hHpi, buffer, &len, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_read() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_hpi_read_single — Read a single 16-bit value from DSP memory using the HPI

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hpi_read_single**(LPVOID hpi, PULONG p_buffer,
ULONG src_addr, int memSpace,
ULONG dspNum);

Description

The pci2040_hpi_read function transfers a 16-bit value from the target DSP memory space to host memory. The HPI accesses DSP memory from the host.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be filled by the read operation. This address must be 32-bit word aligned.
- The *src_addr* parameter is the transfer starting address in the DSP's memory space. This address is from the DSP's point of view.
- The *memSpace* parameter specifies the either program or data space.
- The *dspNum* parameter specifies the DSP to read from.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_read_single() function reads a single 16-bit word from DSP 0's memory at 0xf8.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    value;
   ULONG     l dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(−1);
   }
   if (!pci2040_hpi_read_single(hHpi, &value, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_read_single() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_hpi_write — Write to DSP memory using the HPI

Syntax

```
#include <pci2040dll.h.h>

BOOL pci2040_hpi_write(LPVOID hpi, PULONG p_buffer,
                       PULONG p_length, ULONG dest_addr,
                       int memSpace, ULONG dspNum);
```

Description

The pci2040_hpi_write function transfers data from host memory to the target DSP memory space. The HPI accesses DSP memory from the host.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be transferred by the write operation. This address must be 32-bit word aligned.
- The *p_length* parameter is the address of the write length. The length is updated with the actual transfer length. The length is the number of bytes to be transferred, but must be a multiple of 2 since all transfers are 16-bit words.
- The *dest_addr* parameter is the transfer starting address in the DSP's memory space. This address is from the DSP's point of view.
- The *memSpace* parameter specifies the either program or data space.
- The *dspNum* parameter specifies the DSP to write to.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_write() function writes 200 16-bit words to DSP 0's memory at address 0xf8.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    buffer[1024];
   ULONG     len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
   //fill buffer
   if (!pci2040_hpi_write(hHpi, buffer, &len, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_write() failed}
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_hpi_write_single — Write to a single 16-bit value to DSP memory using the HPI

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hpi_write_single**(LPVOID hpi, PULONG p_buffer,
ULONG dest_addr, int memSpace,
ULONG dspNum);

Description

The pci2040_hpi_write function transfers data from host memory to the target DSP memory space. The HPI accesses DSP memory from the host.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *p_buffer* parameter is the address of the buffer that will be transferred by the write operation. This address must be 32-bit word aligned.
- The *dest_addr* parameter is the transfer starting address in the DSP's memory space. This address is from the DSP's point of view.
- The *memSpace* parameter specifies the either program or data space.
- The *dspNum* parameter specifies the DSP to write to.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_write_single() function writes a single 16-bit word to DSP 0's memory at 0xf8.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    value = 0x1000;
   ULONG     l dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(–1);
   }
   if (!pci2040_hpi_write_single(hHpi, &value, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_write_single() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_hpi_fill — Fill DSP memory using the HPI

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hpi_fill**(LPVOID hpi, USHORT fill_value,
PULONG p_length, ULONG dest_addr,
int memSpace, ULONG dspNum);

Description

The pci2040_hpi_fill function fills target DSP memory space with a fixed data value. The HPI accesses DSP memory from the host.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *fill_value* parameter is the 16-bit data value to be written to DSP memory space.
- The *p_length* parameter is the address of the write length. The length is updated with the actual transfer length. The length is the number of bytes to be transferred, but must be a multiple of 4 since all transfers are 32-bit words.
- The *dest_addr* parameter is the fill starting address in the DSP's memory space. This address is from the DSP's point of view.
- The *memSpace* parameter specifies the either program or data space.
- The *dspNum* parameter specifies the DSP to write to.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hpi_fill() function fills DSP 0's memory starting at address 0xf8 in data memory.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   USHORT    value = 0x1000;
   ULONG     len = 200, dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed  exit(-1);
   }
   if (!pci2040_hpi_fill(hHpi, value, &len, 0x00f8,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_fill() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_generate_int — Generate an interrupt to a DSP

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_generate_int**(LPVOID hpi, PCI2040_DSP_EVENT intr, ULONG dspNum);

Description

The pci2040_generate_int function causes the specified interrupt on the target DSP.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *intr* specifies the type of interrupt to generate.
- The *dspNum* parameter specifies the DSP to interrupt.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_generate_int() function sends a DSP INT3 to the DSP.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   ULONG     dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (hHpi == NULL)
   {
      //HPI open failed
      exit(-1);
   }
   if (!pci2040_generate_int(hHpi, DSP_INT3, dspNum))
   {
      //pci2040_generate_int() failed
   }
   if (!pci2040_hpi_close(hHpi))
   {
      //pci2040_hpi_close() failed
   }
```

pci2040_set_timeout — Set the transfer time-out value

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_set_timeout**(HANDLE hDevice, ULONG timeout);

Description

The pci2040_set_timeout function sets the time-out value for the FIFO data transfers. If a transfer exceeds this time-out period, the transfer will be aborted. The transfer return value will indicate how much of the data was transferred.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *timeout* parameter is the number of milliseconds to wait before terminating a pending transfer. A value of 0 disables the time-out feature. The default value at driver startup is 0.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_set_timeout() function sets the FIFO timeout to 1 second.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_set_timeout(hBoard, 1000))
   {
      //pci2040_set_timeout() failed
   }
```

pci2040_init() Initialize the PCI controller (PCI2040) registers

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_init**(HANDLE hDevice, PPCI2040DLL_INIT_DATA pInitData);

Description

The pci2040_int function configures the PCI controller. A NULL pointer passed into this function results in a default PCI configuration.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pInitData* parameter is a pointer to the PCI configuration to be used.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_init() function initializes the PCI controller.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   PCI2040DLL_INIT_DATA initData;
   hBoard = pci2040_open(board_index, TRUE);
   //set up initData
. . .
   if (!pci2040_init(hBoard, &initData))
   {
      //pci2040_init() failed
   }
```

pci2040_get_configuration() Retrieves the on-board PCI controller configuration

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_get_configuration**(HANDLE hDevice, PPCI2040_DUMP_CONFIG_REGS pReg);

Description

The pci2040_get_configuration function retrieves the PCI controller configuration.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter is a pointer to the PCI configuration register structure that holds the retrieved configuration.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_get_configuration() function retrieves the PCI controller information.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   PCI2040DLL_INIT_DATA initData;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_get_configuration(hBoard, &initData.
      pci_regs))
   {
      //pci2040_get_configuration () failed
   }
```

pci2040_store_configuration() — Writes the on-board PCI controller information

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_store_configuration**(HANDLE hDevice, PPCI2040_DUMP_CONFIG_REGS pReg);

Description

The pci2040_store_configuration function stores the PCI configuration specified by pReg. It is similar to the pci2040_init().

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter is a pointer to the PCI configuration register structure that holds the configuration.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_store_configuration() function updates the PCI controller registers.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   PCI2040DLL_INIT_DATA initData;
   hBoard = pci2040_open(board_index, TRUE);
   //set up initData
. . .
   if (!pci2040_store_configuration(hBoard, &initData.
      pci_regs))
   {
      //pci2040_get_configuration () failed
   }
```

pci2040_enable_int() — Enables a specific interrupt

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_enable_int**(LPVOID hpi, ULONG intMask);

Description

The pci2040_enable_int function enables the DSP-to-HOST or PCI controller interrupt specified by intMask.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *intMask* parameter enumerated in ddpci2040.h specifies the interrupt to enable.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_enable_int() function enables HINT from DSP 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
. . .
   if (!pci2040_enable_int(hBoard, INTDSP0))
   {
      //pci2040_enable_int () failed
   }
```

pci2040_disable_int() — Disables a specific interrupt

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_disable_int**(LPVOID hpi, ULONG intMask);

Description

The pci2040_disable_int function disables the DSP-to-HOST or PCI controller interrupt specified by intMask.

- The *hpi* parameter is the handle returned from a successful pci2040_hpi_open call.
- The *intMask* parameter enumerated in ddpci2040.h specifies the interrupt to enable.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040disable_int() function disables HINT from DSP 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (!pci2040_disable_int(hBoard, INTDSP0))
   {
      //pci2040_disable_int () failed
   }
```

pci2040_hotswap_switch() Returns the status of the hot swap switch

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hotswap_switch**(HANDLE hDevice, PULONG pStat);

Description

The pci2040_hotswap_switch function retrieves the status of the hot swap switch.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pStat* parameter holds the value of the the returned status.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hotswap_switch() function retrieves the hot swap switch status from the PCI controller.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   ULONG     status;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_hotswap_switch (hBoard, &status))
   {
      //pci2040_hotswap_switch () failed
   }
```

pci2040_hotswap_led_on() Turns on the hot swap LED

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hotswap_led_on**(HANDLE);

Description

The pci2040_hotswap_led_on function turns on the hot swap LED.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hotswap_led_on() function turns on the hot swap LED.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   ULONG     status;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_hotswap_led_on (hBoard))
   {
      //pci2040_hotswap_led_on () failed
   }
```

pci2040_hotswap_led_off() — Turns off the hot swap LED

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_hotswap_led_off**(HANDLE hDevice);

Description

The pci2040_hotswap_led_off function turns off the hot swap LED.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_hotswap_led_off() function turns off the hot swap LED.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   LPVOID    hHpi;
   ULONG     status;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_hotswap_led_off (hBoard))
   {
      //pci2040_hotswap_led_off () failed
   }
```

pci2040_printf_control() — Sets the printf redirection function

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_printf_control**(BOOL enable, PPCI2040_PRINTF_FUNC func);

Description

The pci2040_printf_control function sets the function to be used for printf redirection. A typical case is when program output is to be directed to a GUI application.

- The *enable* parameter enables output redirection.
- The *func* parameter is the redirection function to be used.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the function is used to

```
#include <windows.h>
#include <pci2040dll.h>
void myPrintf(char *str);
. . .
pci2040_printf_control (TRUE, myPrinf);
. . .
```

pci2040_input_control() — Sets the input redirection function

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_input_control**(PPCI2040_INPUT_FUNC func);

Description

The pci2040_input_control function sets the function to be used for input redirection. A typical example is when input is coming from a GUI application instead of a DOS terminal.

- The *func* parameter is the redirection function to be used.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the function is used to

```
#include <windows.h>
#include <pci2040dll.h>
void myInput(char *str);
. . .
pci2040_input_control (myInput);
. . .
```

pci2040_dll_revision() — Returns the DLL revision

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_dll_revision**(PULONG pRevMajor,
PULONG pRevMinor,
PULONG pBuildNum);

Description

The pci2040_dll_revision function returns the DLL's revision information.

- The *pRevMajor* parameter is revision major.
- The *pRevMinor* parameter is revision minor.
- The *pBuildNum* parameter is build number.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_dll_revision() function retrieves the DLL revision information

```
#include <windows.h>
#include <pci2040dll.h>
void myPrintf(char *str);
. . .
   ULONG     DllRevMajor;
   ULONG     DllRevMinor;
   ULONG     DllBuildNum;
. . .
   pci2040_dll_revision (&DllRevMajor, &DllRevMinor,
   &DllBuildNum);
```

pci2040_read_reg() — Reads a PCI controller register

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_read_reg**(HANDLE hDevice, PPCI2040_REG pReg)

Description

The pci2040_read_reg returns the PCI controller register value.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter holds the returned value.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_read_reg() function reads a PCI controller register.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int              board_index;
   HANDLE           hBoard;
   PCI2040_REG      reg;
   hBoard = pci2040_open(board_index, TRUE);
   reg.Regoff = 0x54;
   reg.DataVal = 0;
   if (!pci2040_read_reg(hBoard, &reg))
   {
      //pci2040_read_reg() failed
   }
   printf("Value of register at offset: %x = %x.\n",
   reg.RegOff, reg.DataVal);
```

pci2040_write_reg() Writes a PCI controller register

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_write_reg**(HANDLE hDevice, PPCI2040_REG pReg)

Description

The pci2040_write_reg writes a PCI controller register value.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter holds the value to set the register to.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_write_reg() function writes to the PCI controller register at offset 0x5E.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int             board_index;
   HANDLE          hBoard;
   PCI2040_REG     reg;
   hBoard = pci2040_open(board_index, TRUE);
   reg.Regoff = 0x5E;
    reg.DataVal = 0;
   if (!pci2040_read_reg(hDevice, &reg))
      {
         return (FALSE);
      }
   reg.DataVal |= 0x08;
   if (!pci2040_write_reg(hDevice, &reg))
      {
         //pci2040_write_reg() failed
      }
```

pci2040_dsp_present() — Determines if a specific DSP is present

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_dsp_present**(HANDLE hDevice, ULONG dspNum)

Description

The pci2040_dsp_present function determines if the specified DSP is present on the board.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *dspNum* parameter specifies the DSP.

Return Value

The function returns TRUE or FALSE to indicate the presence or absence of the DSP.

Example

In the following example, the pci2040_dsp_present() function determines if the specified DSP is present on the EVM.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   ULONG     dspNum = 0;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_dsp_present(hDevice, dspNum))
   {
      //DSP 0 is absent
   }
   else
   {
      //DSP 0 is present
   }
```

pci2040_mem_config() Sets the DSP memory configuration

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_mem_config**(HANDLE hDevice, ULONG dspNum,
PMemSpaceConfig pMem);

Description

The pci2040_mem_config function sets the memory configuration of the specified DSP. The structure value supplied is not used to write to the DSP but only used to validate memory space accesses.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *dspNum* parameter specifies the DSP.
- The *pMem* parameter indicates the memory configuration.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_mem_config() function reads 200 16-bit words from on-chip SRAM2 at address 0x8000 in data memory on the DSP 0. This read requires the drom bit in the PMST register on the DSP to be set.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int          board_index;
   HANDLE       hBoard;
   LPVOID       hHpi;
   USHORT       buffer[1024];
   LONG         len = 200, dspNum = 0;
   MemSpaceConfig mem;
   mem.ovly = 1;
   mem.drom = 1;
   mem.mp_mc = 0
   hBoard = pci2040_open(board_index, TRUE);
   hHpi = pci2040_hpi_open(hBoard);
   if (!pci2040_mem_config(hBoard, dspNum, &mem))
   {
      //pci2040_mem_config() failed
   }
   if (!pci2040_hpi_read(hHpi, buffer, &len, 0x8000,
      DATA_MEMORY, dspNum))
   {
      //pci2040_hpi_read() failed
   }
   pci2040_hpi_close(hHpi)
```

pci2040_cpld_read_all() — Dumps the CPLD state

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_cpld_read_all**(HANDLE hDevice, PUSHORT pReg)

Description

The pci2040_cpld_read_all function retrieves the CPLD state.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pRegs* parameter is an array of 5 shorts (16-bit values) to store the retrieved state.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_cpld_read_all() function gets a dump of the CPLD register contents.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   USHORT    reg[5];
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_cpld_read_all(hBoard, reg))
   {
      //pci2040_cpld_read_all() failed
   }
```

pci2040_get_event_handle() Returns the handle for a specific event

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_get_event_handle**(HANDLE hDevice,
PCI2040_HOST_EVENT eventNum);

Description

The pci2040_get_event_handle function gets a DSP or PCI controller event handle. This routine is used when the user application wishes to monitor the occurrence of the specified event.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *eventNum* parameter is the event for which the handle is desired.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_get_event_handle() function obtains the handle for the HINT event from DSP 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   HANDLE    hEvent;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_get_event_handle(hBoard, DSP0_INT))
   {
      //pci2040_get_event_handle() failed
   }
```

pci2040_read_cpld_reg() — Reads a CPLD controller register

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_read_cpld_reg**(HANDLE hDevice, PPCI2040_REG pReg);

Description

The pci2040_read_cpld_reg function reads the value of a CPLD register.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter holds the value of the returned register value. The register offset to be used for the pReg pointer is defined in pci2040dll.h.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_read_cpld_reg() function reads a specific CPLD register.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int             board_index;
   HANDLE          hBoard;
   PCI2040_REG     reg;
   hBoard = pci2040_open(board_index, TRUE);
   reg.Regoff = 0x02;
   reg.DataVal = 0;
   if (!pci2040_read_cpld_reg(hBoard, &reg))
   {
      //pci2040_read_cpld_reg() failed
   }
   printf("Value of register at offset: %x = %x.\n",
          reg.RegOff, reg.DataVal);
```

pci2040_write_cpld_reg() Writes a test bus controller register

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_write_cpld_reg**(HANDLE hDevice, PPCI2040_REG pReg);

Description

The pci2040_write_cpld_reg function reads the value of CPLD register.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *pReg* parameter holds the value to write to the register. The register offset to be used for the pReg pointer is defined in pci2040dll.h.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_write_cpld_reg() function writes to specific CPLD register.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int             board_index;
   HANDLE          hBoard;
   PCI2040_REG     reg;
   hBoard = pci2040_open(board_index, TRUE);
   reg.Regoff = 0x02;
   reg.DataVal = 0x01;
   pci2040_read_cpld_reg(hBoard, &reg);
   reg.DataVal |= 0x01;
   if (!pci2040_write_cpld_reg(hBoard, &reg))
   {
      //pci2040_write_reg() failed
   }
```

pci2040_user_semaphore_wait() Waits indefinitely for a semaphore

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_user_semaphore_wait**(HANDLE hDevice,
PCI2040_SEM sem);

Description

The pci2040_user_semaphore_wait function waits indefinitely to acquire the specified user semaphore.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *sem* parameter indicates the semaphore to acquire.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_user_semaphore_wait() function waits indefinitely for CPLD semaphore 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_user_semaphore_wait (hBoard,
      PCI2040_SEM0))
   {
      //pci2040_user_semaphore_wait() failed
   }
```

pci2040_user_semaphore_get() — Acquires a semaphore

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_user_semaphore_get**(HANDLE hDevice,
PCI2040_SEM sem);

Description

The pci2040_user_semaphore_wait function tries to acquire the specified user semaphore. This routine returns immediately with a FALSE if the semaphore is not available.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *sem* parameter indicates the semaphore to acquire.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_user_semaphore_get() function acquires CPLD semaphore 0.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_user_semaphore_get (hBoard, PCI2040_SEM0))
   {
      //pci2040_user_semaphore_get() failed
   }
```

pci2040_user_semaphore_release() Releases a semaphore

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_user_semaphore_release(**HANDLE hDevice,
PCI2040_SEM sem);

Description

The pci2040_user_semaphore_release function releases a semaphore that was previously acquired by the host.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *sem* parameter indicates the semaphore to release.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_user_semaphore_release() function releases a previously acquired CPLD semaphore.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
   if (!pci2040_user_semaphore_get (hBoard, PCI2040_SEM0))
   {
      //pci2040_user_semaphore_get() failed
   }
. . .
   if (!pci2040_user_semaphore_release (hBoard,
      PCI2040_SEM0))
   {
      //pci2040_user_semaphore_release() failed
   }
```

pci2040_get_base_address() — Returns a base address

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_get_base_address(**HANDLE hDevice, PCI2040_BAR barNum);

Description

The pci2040_get_base_address function returns the board device’s base address value specified by bar_num.

- The *hDevice* parameter is the handle returned from a successful pci2040_open call.
- The *barNum* parameter indicates the base address to obtain.

Return Value

The function returns the base address value upon success or NULL upon failure.

Example

In the following example, the pci2040_get_base_address() function obtains the HPI base address.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   PULONG    hpiBar = NULL;
   hBoard = pci2040_open(board_index, TRUE);
   hpiBar = pci2040_get_base_address (hBoard, HPI_BAR);
   if (hpiBar == NULL)
   {
      //pci2040_get_base_address() failed
   }
```

pci2040_tbc_reset() — Reset the Test Bus Controller

Syntax

#include <pci2040dll.h.h>

BOOL **pci2040_tbc_reset(**HANDLE hDevice);

Description

The pci2040_tbc_reset function resets the test bus controller. All the counter registers are set to zero.

- ❑ The *hDevice* parameter is the handle returned from a successful pci2040_open call.

Return Value

The function returns TRUE or FALSE to indicate the success of the operation.

Example

In the following example, the pci2040_tbc_reset () function resets the TBC.

```
#include <windows.h>
#include <pci2040dll.h>
. . .
   int       board_index;
   HANDLE    hBoard;
   hBoard = pci2040_open(board_index, TRUE);
. . .
   if (!pci2040_tbc_reset (hBoard))
   {
      //pci2040_tbc_reset() failed
   }
```

Chapter 5

PCI2040 EVM DSP Support Software

This chapter describes the EVM DSP board support software by providing APIs and example code for the audio codec library. This module uses the TMS320C5410 device library to access and control internal peripheral registers.

Topic **Page**

5.1 DSP Support Software Components 5-2

5.2 Using the DSP Support Software Components 5-2

5.3 Board Driver API 5-3

5.4 CODEC Library API 5-27

5.1 DSP Support Software Components

The DSP support software consists of the codec library as well as other useful board APIs. The example code provided operates the TLC320AD77C audio codec in serial data mode and all audio data is communicated to and from the ’C5410 via the McBSP port 1. Configuration of the serial port is completely hidden from the user and as such the user only needs to make calls to the codec library to facilitate codec operation.

The codec library is a collection of routines which configure and control the operation of the TLC320AD77C audio codec. The API functions correspond closely to the functional organization of the chip. Macros are defined for this library in the file codec.h which should be used as arguments to the API functions.

5.2 Using the DSP Support Software Components

The DSP support software, which consists of the audio codec library and the board support library, is installed from the accompanying CD-ROM to the \pci2040\dsp\lib directory. These components are in object format and are supplied in the archived object library file board.lib. The source for the DSP support files is contained in the source libray file board.src. To extract the source file, assuming you have installed the ’C54x code generation tools, enter:

```
ar500 –x board.src
```

This command extracts the makefile in the board.src file. The DSPPATH variable in the makefile must be modified to point to the c54xtools directory.

To build the object files from the extracted source (*.c) files, change to the directory containing the files and enter:

```
make
```

GNU make is required for this to work. Examples code that uses the audio codec library and board support library exists in the .\dsp\examples directory. You can run the example code via the ’C54x EVM debugger, or you can load the code and allow it to run using the EVM COFF loader utility (evmload.exe). The print statements that are visible in the debugger command window are not visible on the DOS screen when you use the the COFF loader.

5.3 Board Driver API

This section discusses the board API. Included in this discussion are the macros, data types, and functions defined which comprise the board driver for the PCI2040 TMS320C54x EVM board.

5.3.1 Board Driver Macros

None

5.3.2 Board Driver Data Types

This section lists the public data types defined by the board driver.

- CPLD semaphores

```
typedef enum
{
   EVM_SEM0,
   EVM_SEM1
} sem_sel;
```

This type is required to use the semaphore access routines.

- Host interrupts

```
typedef enum
{
   EVM_HINT,
   EVM_HINT2
} EvmInt;
```

This enumerates the various interrupts that can be used to interrupt the host.

5.3.3 Board Driver Functions

Function	Description
evm_int()	Initialize EVM board
evm_interrupt_host()	Interrupt host
evm_led_enable()	Enable user LED
evm_led_disable()	Disable user LED
evm_toggle_led()	Toggle user LED
evm_sem_get()	Acquire a CPLD semaphore
evm_sem_wait()	Wait on a CPLD semaphore
evm_sem_release()	Release a CPLD semaphore
evm_fifo_read()	Read from FIFO
evm_fifo_write()	Write to FIFO
evm_delay_msec()	Delay CPU for specified number of milliseconds
evm_delay_usec()	Delay CPU for specified number of microseconds
evm_set_wait_states()	Set wait states for memory spaces
evm_set_cpu_freq()	Set the CPU frequency
evm_get_cpu_freq()	Get the CPU frequency
evm_io_redir()	Redirect I/O writes to host terminal
evm_mem_transfer()	Transfer data using a specific DMA channel
cio_driver_init()	Initialize terminal I/O
cio_open()	Open terminal I/O channel
cio_close()	Close terminal I/O channel
cio_read()	Read from I/O channel
cio_write()	Write to I/O channel

evm_int() — Initialize EVM board

Syntax

#include < board.h>

int **evm_int**(int cpuFreq);

Defined in

board.c as a callable C routine

Description

This function initializes the board. It sets the CPU frequency, sets the CLKOUT to CLKIN/2, initializes the CPLD registers, and enables global interrupts.

- The *cpuFreq* parameter selects the frequency to run the board at.

Return Value

OK if successful, ***ERROR*** otherwise.

Example

Initialize the board to run at 100 MHz.

```
int status;
unsigned int cpu_freq = 100;
status = evm_init(cpu_freq);
if (status == ERROR)
   return(ERROR);
```

evm_interrupt_host() — Interrupt host

Syntax

```
#include < board.h>

int evm_interrupt_host(EvmInt intr);
```

Defined in board.c as a callable C routine

Description This function generates an interrupt to the host.

Return Value ***OK*** if successful, ***ERROR*** otherwise.

evm_led_enable() — Enable user LED

Syntax

#include < board.h>

int **evm_led_enable**();

Defined in

board.c as a callable C routine

Description

This function turns on the user LED on the EVM board bracket.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_led_disable() — Disable user LED

Syntax

#include < board.h>

int **evm_led_disable**();

Defined in

board.c as a callable C routine

Description

This function turns off the user LED on the EVM board bracket.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_toggle_led() — Toggle user LED

Syntax

#include < board.h>

int **evm_toggle_led**();

Defined in

board.c as a callable C routine

Description

This function toggles the user LED on the EVM board bracket.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_sem_get() — Acquire a CPLD semaphore

Syntax

#include < board.h>

int **evm_sem_get**();

Defined in

board.c as a callable C routine

Description

This function tries to acquire the specified CPLD semaphore. The function returns immediately with an error if the semaphore is not available.

- The *sem* parameter selects the semaphore to acquire

Return Value

OK if successful, ***ERROR*** otherwise.

evm_sem_wait() — Wait for a CPLD semaphore

Syntax

#include < board.h>

int **evm_sem_wait**(sem_sel sem);

Defined in

board.c as a callable C routine

Description

This function waits indefinitely to acquire the specified CPLD semaphore.

- The *sem* parameter selects the semaphore to acquire

Return Value

OK if successful, ***ERROR*** otherwise.

evm_sem_release() — Release a CPLD semaphore

Syntax

#include < board.h>

int **evm_sem_release**(sem_sel sem);

Defined in

board.c as a callable C routine

Description

This function releases a CPLD semaphore previously acquired by the board.

- The *sem* parameter selects the semaphore to release

Return Value

OK if successful, ***ERROR*** otherwise.

evm_fifo_read() — Read from FIFO

Syntax

#include < board.h>

int **evm_fifo_read** (int fildes, char *bufPtr, unsigned int cnt);

Defined in

board.c as a callable C routine

Description

This function reads *cnt* number of words from the FIFO. If the FIFO is not busy, then this call blocks until the required number of words are read. Otherwise, it returns immediately.

- The *fildes* parameter is the file descriptor. This parameter is not used internally and therefore can be set to 0.
- The *bufPtr* parameter is a pointer to the buffer to store the data.
- The *cnt* parameter is the number of words to read from the FIFO.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_fifo_write() Write to FIFO

Syntax

#include < board.h>

int **evm_fifo_write** (int fildes, char *bufPtr, unsigned int cnt);

Defined in

board.c as a callable C routine

Description

This function writes *cnt* number of words to the FIFO. If the FIFO is not busy, then this call blocks until the required number of words are written to the FIFO. Otherwise, it returns immediately.

- The *fildes* parameter is the file decsriptor. This parameter is not used internally and therefore can be set to 0.
- The *bufPtr* parameter is a pointer to the buffer to store the data.
- The *cnt* parameter is the number of words to read from the FIFO.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_delay_msec() Delay CPU for specified number of milliseconds

Syntax

#include < board.h>

int **evm_delay_msec** (unsigned int msec);

Defined in

board.c as a callable C routine

Description

This function delays the CPU for the specified number of milliseconds.

- The *msec* parameter is the number of milliseconds to delay the CPU.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_delay_usec() Delay CPU for specified number of microseconds

Syntax

#include < board.h>

int **evm_delay_usec** (unsigned int usec);

Defined in

board.c as a callable C routine

Description

This function delays the CPU for the specified number of microseconds.

- The *usec* parameter is the number of microseconds to delay the CPU.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_set_wait_states() — Set wait states for memory spaces

Syntax

```
#include < board.h>

int evm_set_wait_states (unsigned int extProgWait,
                         unsigned int dbProgWait,
                         unsigned int dataWait,
                         unsigned int ioWait);
```

Defined in

board.c as a callable C routine

Description

This function sets the wait states for the various memory spaces.

- The *extProgWait* parameter specifies the number of external program memory wait states.
- The *dbProgWait* parameter specifies the number of daughterboard memory wait states.
- The *dataWait* parameter specifies the number of data memory wait states.
- The *ioWait* parameter specifies the number of IO memory wait states.

Return Value

OK if successful, ***ERROR*** otherwise.

evm_set_cpu_freq() — Set the CPU frequency

Syntax

#include < board.h>

void **evm_set_cpu_freq** (unsigned int cpuFreq);

Defined in

board.c as a callable C routine

Description

This function sets the operating CPU frequency.

- The *cpuFreq* parameter specifies the frequency to operate the CPU at.

Return Value

None

evm_get_cpu_freq() Get the CPU frequency

Syntax #include < board.h>

unsigned int **evm_get_cpu_freq** ();

Defined in board.c as a callable C routine

Description This function retrieves the CPU operating frequency.

Return Value ***CPU* frequency**

evm_io_redir() — Redirect I/O writes to host terminal

Syntax

```
#include < board.h>

void evm_io_redir ();
```

Defined in

board.c as a callable C routine

Description

This function redirects I/O (printf, etc) to the host terminal. Once this call is made, terminal I/O can be enabled or disabled using dip switch 6 on the EVM board.

Return Value

None

evm_mem_transfer() Transfer data using a specific DMA channel

Syntax

```
#include < board.h>

int evm_mem_transfer (unsigned int ch, unsigned int *src_addr,
                      unsigned int src_space, unsigned int *dst_addr,
                      unsigned int dst_space, unsigned int cnt, bool sync);
```

Defined in

board.c as a callable C routine

Description

This function transfers data from one memory to another.

- The *ch* parameter specifies the DMA channel to use.
- The *src_addr* parameter specifies the source address of the data to transfer.
- The *src_space* parameter specifies the memory space of the source data. The correct values can be found in the ’C54x device library header file dma5410.h
- The *dst_addr* parameter specifies the destination address.
- The *dst_space* parameter specifies memory space for the destination address.
- The *cnt* parameter is the number of words to transfer.
- The *sync* parameter indicates if you want to block until transfer is complete or if the function should return right after initiating transfer.

Return Value

OK if successful, ***ERROR*** otherwise.

cio_driver_init() — Initialize terminal I/O

Syntax

#include < board.h>

void **cio_driver_init** ();

Defined in

board.c as a callable C routine

Description

This function initializes internal data structures used by the FIFO and terminal I/O routines.

Return Value

None

cio_open() — Open terminal I/O channel

Syntax

#include < board.h>

int **cio_open** ();

Defined in

board.c as a callable C routine

Description

This function opens the terminal I/O channel. Applications have to make a call to this function prior to calling either cio_read() or cio_write().

Return Value

int – A file descriptor or ***ERROR*** if cio_open() failed.

cio_close() — Close terminal I/O channel

Syntax

#include < board.h>

int **cio_close**(int fildes);

Defined in

board.c as a callable C routine

Description

This function closes a previously opened terminal I/O channel.

- The *fildes* parameter specifies the file descriptor resulting from a previous call to cio_open.

Return Value

OK if successful, ***ERROR*** otherwise.

cio_read() — Read from I/O channel

Syntax

#include < board.h>

int **cio_read** (int fildes, char *bufPtr, unsigned int cnt);

Defined in

board.c as a callable C routine

Description

This function reads from the terminal I/O channel. If the channel is not busy, then this call blocks until the required number of words are read. Otherwise, it returns immediately.

- The *fildes* parameter specifies the file descriptor resulting from a previous call to cio_open. This parameter can only be 0, 1, or 2.
- The *bufPtr* parameter is the buffer to store the data.
- The C*nt* parameter is the number of words to read.

Return Value

int – The number of words read.

cio_write() — Write to I/O channel

Syntax

#include < board.h>

int **cio_write**(int fildes, char *bufPtr, unsigned int cnt);

Defined in

board.c as a callable C routine

Description

This function writes to the terminal I/O channel. If the channel is not busy, then this call blocks until the required number of words are written. Otherwise, it returns immediately.

- ❑ The *fildes* parameter specifies the file descriptor resulting from a previous call to cio_open. This parameter can only be 0, 1, or 2.
- ❑ The *bufPtr* parameter is the buffer containing the data to write.
- ❑ The C*nt* parameter is the number of words to write.

Return Value

int – The number of words read.

5.4 CODEC Library API

This section discusses the audio CODEC library API. Included in this discussion are the macros and functions defined which comprise the audio CODEC library for the PCI2040 EVM board.

5.4.1 CODEC Library Macros

Table 5–1 lists the macros defined, their values, and a description of each. The API functions which use each set of macros are also listed. These macros are defined within the public header file codec.h.

Table 5–1. Mode and Input Selection Macros

a) Mode selection macros

Macro	Value	Description
PLAY	0	Selects playback mode for control
CAPTURE	1	Selects capture mode for control.
Used by :	codec_reload( )	

b) Input Selection Macros

Macro	Value	Description
LINE_SEL	0	Selects the line input.
MIC_SEL	1	Selects the microphone input
Used by :	codec_input( )	

5.4.2 CODEC Data Types and Type Definitions

(a) Codec De-emphasis modes

```
typedef enum
{
   D_32000 = 0x00,              /* 32000 Hz */
   D_44100 = 0x01,              /* 44100 Hz */
   D_48000 = 0x02,              /* 48000 Hz */
   D_OFF = 0x03                 /* OFF  Hz */
} dem_sel;
```

Specifies the de-emphasis mode for the codec.

(b) Codec sampling rates

```
typedef enum
{
   F_8000 = 0x0014,            // 8000 Hz
   F_9600 = 0x0010,            // 9600 Hz
   F_11025 = 0x0094,           // 11025 Hz
   F_12000 = 0x000c,           // 12000 Hz
   F_13230 = 0x0090,           // 13230 Hz
   F_16000 = 0x0008,           // 16000 Hz
   F_16537_5 = 0x008c,         // 16537.5 Hz
   F_22050 = 0x0088,           // 22050 Hz
   F_24000 = 0x0004,           // 24000 Hz
   F_32000 = 0x0003,           // 32000 Hz
   F_33075 = 0x0084,           // 33075 Hz
   F_44100 = 0x0083,           // 44100 Hz
   F_48000 = 0x0002,           // 48000 Hz
   F_66150 = 0x0082,           // 66150 Hz
   F_96000 = 0x0001            // 96000 Hz
} c_srate;
```

5.4.3 CODEC Library Functions

Function	Description
codec_init()	Initializes the codec
codec_reset()	Resets the codec
codec_sample_rate()	Sets the sampling rate on the codec
codec_enumToHz()	Converts the codec sampling rate enumeration to a floating point value
codec_de_emphasis()	Sets the codec de-emphasis mode
codec_input()	Selects the input source for the code
codec_cont_play()	Continously plays supplied buffers
codec_cont_capture()	Continuously capture data into supplied buffers
codec_single_play()	Play supplied buufer
codec_single_capture()	Fill supplied buffer with data
codece_reload_setup()	Sets up reload parameters for the codec

codec_init() — Initializes the CODEC

Syntax

#include < codec.h>

int **codec_init** ();

Defined in

codec.c as a callable C routine

Description

This function initializes, sets default parameters and calibrates the TLC320AD77C CODEC.

Return Value

OK if successful, ***ERROR*** otherwise.

Example

Initialize the CODEC for use.

```
int status;
status= codec_init( );
if (status == ERROR)
   return(ERROR);
```

codec_reset() — Resets the codec

Syntax

#include < codec.h>

int **codec_reset** ();

Defined in

codec.c as a callable C routine

Description

This function resets the TLC320AD77C CODEC. It stops any ongoing transfer and initializes its internal state.

Return Value

None.

codec_sample_rate() — Sets the sampling rate on the codec

Syntax

#include < codec.h>

int **codec_sample_rate** (c_srate sampRate);

Defined in

codec.c as a callable C routine

Description

This function sets the sampling rate of the codec. This call powers down the codec before setting the sampling rate so care must be taken not to use this call to change the codec’s sampling rate while its playing data.

- The *c_srate* parameter specifies the sampling rate to which the codec is set.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_enumToHz() — Converts the codec sampling rate enumeration to floating point

Syntax

#include < codec.h>

int **codec_enumToHz** (c_srate sampRate);

Defined in

codec.c as a callable C routine

Description

This function returns the floating point representation of the codec sampling rate.

- The *c_srate* parameter specifies the sampling rate to which the codec is set.

Return Value

Floating point value of sampling rate or 0 for an unsupported sampling rate.

codec_de_emphasis() — Sets the codec de-emphasis mode

Syntax

#include < codec.h>

int **codec_de_emphasis** (dem_sel dem);

Defined in

codec.c as a callable C routine

Description

This function sets the de-emphasis mode for the codec.

- The *dem* specifies parameter de-emphasis to use.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_input() Selects the input source for the code

Syntax

```
#include < codec.h>

int codec_input (int left, int right);
```

Defined in

codec.c as a callable C routine

Description

This function selects the input source of the codec.

- The *left* parameter specifies the left channel input source.
- The *right* parameter specifies the right channel input source.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_cont_play() — Continously plays supplied buffers

Syntax

#include < codec.h>

int **codec_cont_play** (int *pingBuf, int *pongBuf, unsigned int size,
Fp callBack);

Defined in

codec.c as a callable C routine

Description

This function continuously plays the data in the supplied ping-pong buffers. The optional callback routine is called at the end of each buffer. This call uses the DMA controller for the transfer and as such is nonblocking.

- The *pingBuf* parameter specifies the first buffer to be played.
- The *pongBuf* parameter specifies the other buffer.
- The *size* parameter specifies the number of words 16-bit words in the buffer to play.
- The *callBack* parameter specifies the function to call at the end of each buffer.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_cont_capture() Continuously capture data into supplied buffers

Syntax

#include < codec.h>

int **codec_cont_capture** (int *pingBuf, int *pongBuf, unsigned int size, Fp callBack);

Defined in

codec.c as a callable C routine

Description

This function continuously captures data into the supplied ping-pong buffers. The optional callback routine is called at the end of each buffer. This call uses the DMA controller for the transfer and as such is nonblocking.

- The *pingBuf* parameter specifies the first buffer to fill.
- The *pongBuf* parameter specifies the other buffer.
- The *size* parameter specifies the number of 16-bit words to put in the buffer.
- The *callBack* parameter specifies the function to call when each buffer is full.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_single_play() — Play supplied buffer

Syntax

#include < codec.h>

int **codec_singl_play** (int* buffer, unsigned int size, Fp callBack);

Defined in

codec.c as a callable C routine

Description

This function plays the data in the supplied buffer. The optional callback routine is called at the end of the buffer. This call uses the DMA controller for the transfer and as such is nonblocking.

- The *buf* parameter specifies the buffer to be played.
- The *size* parameter specifies the number of 16-bit words in the buffer to play.
- The *callBack* parameter specifies the function to call at the end of the buffer.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_single_capture() — Fill supplied buffer with data

Syntax

#include < codec.h>

int **codec_single_capture** (int *buf, unsigned int size, Fp callBack);

Defined in

codec.c as a callable C routine

Description

This function continuously captures data into the supplied buffer. The optional callback routine is called at the end of the buffer. This call uses the DMA controller for the transfer and as such is nonblocking.

- The *buf* parameter specifies the buffer to fill.
- The *size* parameter specifies the number of 16-bit words to put in the buffer.
- The *callBack* parameter specifies the function to call when the buffer is full.

Return Value

OK if successful, ***ERROR*** otherwise.

codec_reload_setup() Sets up reload parameters for the codec

Syntax

#include < codec.h>

int **codec_reload** (unsigned int mode, int *src, unsigned int *dst,
unsigned int size, bool start);

Defined in

codec.c as a callable C routine

Description

This function sets up the internal DMA registers ready for the next transfer. This call may be use in conjunction with codec_single_play() or codec_single_capture() to simulate a continuous play or capture.

- The *mode* parameter specifies PLAY or CAPTURE
- The *src* parameter specifies the source address. If mode is equal to capture, then this value is ignored.
- The *dst* parameter specifies the destination address. If mode is equal to play, then this value is ignored.
- The *size* parameter specifies the number of 16-bit words to put in the buffer.
- The *start* parameter tells the function to start transfer right after reloading the DMA registers.

Return Value

OK if successful, ***ERROR*** otherwise.

5.4.4 Sample Code

```
#include <stdio.h>
#include <stdlib.h>
#include <file.h>
#include <float.h>
#include <math.h>
#include <string.h>
#include <type.h>
#include <common.h>
#include <test_sup.h>
#include <_board.h>
#include <intr.h>
#include <board.h>
#include <codec.h>
#include <mcbsp54.h>
#define PRINT_DBG     0
/*********************************************************/
/* defines                                               */
/*********************************************************/
#define LEFT       0
#define RIGHT      1
#define BOTH       2
#define EMU_CNTL   0
#define INTEGER_PART              0x7f80
#define VOICE_BAND_SAMPLE_RATE    F_16000
#define PRO_AUDIO_SAMPLE_RATE     F_44100
#define TONE_PB_SAMPLE_RATE       F_44100
#define SAMPLE_RATE               F_16000
#define BUF_SIZE                  1024  // 1k buffers
#define TABLE_SIZE                256
#define NUM_BUFFS_TO_PLAY         100
//used to generate sine wave using a table lookup
typedef struct {
   u16 offset;
   u16 delta;
} Sine, PSine;
/*********************************************************/
/* globals data                                          */
/*********************************************************/
Sine  sig = {0, 0};
bool  ind = 0;
s16   toneBuf[2][BUF_SIZE];
s16   *pBuf[2];
u32   ch;
bool  isr_ind = 0;
bool  local_index = 0;
bool  bufFlag[2] = {1,1};
u16   bufsPlayed = 0;
bool  codecOn;
bool  stop = False;
u16   toneFreq = 3100;
```

```
/*******************************************************/
/* 256 point sinusoidal signal                          */
/*******************************************************/
const s16 sine_table[TABLE_SIZE] = {
      804,  1608,  2411,  3212,  4011,  4808,  5602,  6393,
     7180,  7962,  8740,  9512, 10279, 11039, 11793, 12540,
    13279, 14010, 14733, 15447, 16151, 16846, 17531, 18205,
    18868, 19520, 20160, 20788, 21403, 22006, 22595, 23170,
    23732, 24279, 24812, 25330, 25833, 26320, 26791, 27246,
    27684, 28106, 28511, 28899, 29269, 29622, 29957, 30274,
    30572, 30853, 31114, 31357, 31581, 31786, 31972, 32138,
    32286, 32413, 32522, 32610, 32679, 32729, 32758, 32767,
    32758, 32729, 32679, 32610, 32522, 32413, 32286, 32138,
    31972, 31786, 31581, 31357, 31114, 30853, 30572, 30274,
    29957, 29622, 29269, 28899, 28511, 28106, 27684, 27246,
    26791, 26320, 25833, 25330, 24812, 24279, 23732, 23170,
    22595, 22006, 21403, 20788, 20160, 19520, 18868, 18205,
    17531, 16846, 16151, 15447, 14733, 14010, 13279, 12540,
    11793, 11039, 10279,  9512,  8740,  7962,  7180,  6393,
     5602,  4808,  4011,  3212,  2411,  1608,   804,     0,
     -804, -1608, -2411, -3212, -4011, -4808, -5602, -6393,
    -7180,   -7962,   -8740,   -9512,   -10279,  -11039,
    -11793,  -12540,
    -13279,  -14010,  -14733,  -15447,  -16151,  -16846,
    -17531,  -18205,
    -18868,  -19520,  -20160,  -20788,  -21403,  -22006,
    -22595,  -23170,
    -23732,  -24279,  -24812,  -25330,  -25833,  -26320,
    -26791,  -27246,
    -27684,  -28106,  -28511,  -28899,  -29269,  -29622,
    -29957,  -30274,
    -30572,  -30853,  -31114,  -31357,  -31581,  -31786,
    -31972,  -32138,
    -32286,  -32413,  -32522,  -32610,  -32679,  -32729,
    -32758,  -32768,
    -32758,  -32729,  -32679,  -32610,  -32522,  -32413,
    -32286,  -32138,
    -31972,  -31786,  -31581,  -31357,  -31114,  -30853,
    -30572,  -30274,
    -29957,  -29622,  -29269,  -28899,  -28511,  -28106,
    -27684,  -27246,
    -26791,  -26320,  -25833,  -25330,  -24812,  -24279,
    -23732,  -23170,
    -22595,  -22006,  -21403,  -20788,  -20160,  -19520,
    -18868,  -18205,
    -17531,  -16846,  -16151,  -15447,  -14733,  -14010,
    -13279,  -12540,
    -11793,  -11039,  -10279,  -9512,   -8740,   -7962,
    -7180,   -6393,
    -5602, -4808, -4011, -3212, -2411, -1608,  -804,     0
};
/*******************************************************/
/* functions                                            */
/*******************************************************/
```

```
static void genTone(s16 *pBuf, u16 size, u16 ch);
static s16 CodecExLb(s16 micSel, int mode);
static s16 CodecExContTonePb( s16 ch, s16 duration );
static void ContTonePbCallback(void);
static void SingleTonePbCallback(void);
static int CodecExBlkCapPb(int  micSel, int mode);
static s16 CodecExContTonePbUseIsr( s16 ch, s16 duration);
interrupt void _serialRxIsr(void);
interrupt void _tempIsr();
/*********************************************************/
/* MAIN                                                  */
/*********************************************************/
main()
{
   s16 command[2];
   s16 reply;
   s16 done = False;
   s16 i=2;
if (evm_init(100))
   return;

#if EMU_CNTL
   // blink the leds a couple times
   while ( i-- )
   {
      evm_toggle_led();
      evm_delay_msec(500);
   }
#endif
   sig.delta = (u16)( ( (volatile f32)toneFreq /
                codec_enumToHz(SAMPLE_RATE) ) * 256);
   sig.delta <<=  7;
   /****************************************************/
   /* Codec Tone Generator using DMA autoinitialization */
   /* example                                           */
   /****************************************************/
#if 1
   printf("Codec Continuous Tone Playback example.   \n");
   if (CodecExContTonePb(BOTH, NUM_BUFFS_TO_PLAY ))
   {
      return(ERROR);
   }
#endif
   /****************************************************/
   /* Codec tone generator using serial interrupts      */
   /* example                                           */
   /****************************************************/
#if 1
   printf("Codec Continuous Tone Playback example using
           serial interrupts. \n");
   if (CodecExContTonePbUseIsr(BOTH, 3 ))
   {
      return(ERROR);
```

```
   }
#endif
   /****************************************************/
   /* Codec loopback example                       */
   /* Codec simultaneous capture and playback over      */
   /* McBSP example                                     */
   /****************************************************/
#if 1
   printf("\nCodec Line In Loopback example.   \n");
   if (CodecExLb(MIC_SEL, WAIT_FOR_INPUT))
   {
      return(ERROR);
   }
#endif
#if 1
   /****************************************************/
   /* Block Capture and Playback Example                */
   /****************************************************/
   printf("Codec Block Capture Playback example.   \n");
   if (CodecExBlkCapPb(MIC_SEL, WAIT_FOR_INPUT) == ERROR)
   {
      //printf("Error calling CodecExBlkCapPb()\n  ");
      return(ERROR);
   }
#endif
   printf("Thats All Folks\n");
   return(OK);
}
/*******************************************************/
/* CodecExContTonePbUseIsr – CODEC Example: Isr mode    */
/*                                                     */
/* This function provides an example of using the      */
/* TLC320AD77C CODEC in Loopback Mode. Input is looped  */
/* back to the output                                  */
/*******************************************************/
static s16 CodecExContTonePbUseIsr( s16 ch, s16 duration )
{
   u16 cnt = duration;
      if (codec_init())
         return (ERROR);
      sig.delta = (u16)( ( (volatile f32)toneFreq /
                   codec_enumToHz(SAMPLE_RATE) ) * 256);
      sig.delta <<=  7;
      codec_sample_rate(SAMPLE_RATE);
      INTR_INIT;
      INTR_HOOK(XINT1_TRAP, _tempIsr);
      INTR_ENABLE(XINT1);
      evm_mcbsp_enable(1,2);
      while (cnt--)
   evm_delay_msec(5000);
   MCBSP_TX_RESET(1);
   return OK;
```

```
}
interrupt void _tempIsr()
{
   s16 scaledVal;
      scaledVal = sine_table[ ((sig.offset + 0x0040) &
                  INTEGER_PART) >> 7 ] >> 1;
      sig.offset += sig.delta;
      *(u16 *)(DXR1_ADDR(1)) = scaledVal;
      *(u16 *)(DXR2_ADDR(1)) = scaledVal;
}
/*********************************************************/
/* CodecExContTonePb() – Play continuous tone           */
/* This function provides an example of playing back a  */
/* buffer of data in continuous (dma autoinitiali-      */
/* zation) mode.                                         */
/*********************************************************/
static s16 CodecExContTonePb( s16 ch, s16 duration )
{
   evm_led_enable();
   sig.delta = (u16)( ( (volatile f32)toneFreq /
            codec_enumToHz(TONE_PB_SAMPLE_RATE) ) * 256);
   sig.delta <<=  7;
   if (codec_init())
   {
#if PRINT_DBG
   return (ERROR);
#endif
   }
   codec_sample_rate(TONE_PB_SAMPLE_RATE);
   //synthesize tone
   genTone(toneBuf[0], BUF_SIZE, ch);
   genTone(toneBuf[1], BUF_SIZE, ch);
   bufsPlayed = 0;
   codecOn = True;
   if (codec_cont_play(toneBuf[0], toneBuf[1], BUF_SIZE,
        ContTonePbCallback) == ERROR)
   {
#if PRINT_DBG
      printf("Error playing buffer to codec\n");
#endif
      return (ERROR);
   }
   isr_ind = local_index = 0;
   while( bufsPlayed < duration)
   {
      //wait for buffer to be empty
      while (bufFlag[local_index]);
      //fill buffer and setup for next buffer
      genTone(toneBuf[local_index], BUF_SIZE, ch);
      bufFlag[local_index] = 1;
      local_index ^= 1;
   }
   codecOn = False;
```

```
   MCBSP_TX_RESET(1);
   codec_reset();
   evm_led_disable();
   return(OK);
}
/*******************************************************/
/* ContTonePbCallback() – Codec continuous play         */
/* callback routine.                                   */
/*******************************************************/
static void ContTonePbCallback(void)
{
   bufsPlayed++;
   bufFlag[isr_ind] = 0;
   while ((!bufFlag[isr_ind ^= 1]) && codecOn);
}
/*******************************************************/
/* CodecExLb –  CODEC Example: Loopback mode           */
/* This function provides an example of using the      */
/* TLC320AD77C CODEC in Digital Loopback Mode (DLB).   */
/*******************************************************/
static s16 CodecExLb(s16 micSel, int mode)
{
u16 cnt = 4;
   evm_led_enable();
   if (codec_init())
      return(ERROR);
   if (micSel)
   {
      if (codec_input(MIC_SEL, MIC_SEL))
         return(ERROR);
      if (codec_sample_rate(VOICE_BAND_SAMPLE_RATE))
         return(ERROR);
#if PRINT_DBG
   printf("\nBegin speaking into the microphone when the
          LED illuminates. \n");
#endif
   }
   else
   {
      if (codec_input(LINE_SEL, LINE_SEL))
         return(ERROR);
      if (codec_sample_rate(PRO_AUDIO_SAMPLE_RATE))
         return(ERROR);
#if PRINT_DBG
   printf("\n<Play input source when the LED
          illuminates.>\n");
#endif
   }
   codec_reset();
   INTR_INIT;
   INTR_HOOK(RINT1_TRAP, _serialRxIsr);
   INTR_ENABLE(RINT1);
   evm_led_enable();
```

```
   //enable mcbsp 1 serial port tx and rx
   evm_mcbsp_enable(1,3);
while (cnt--)
   evm_delay_msec( 5000 );;
   MCBSP_RX_RESET(1);
   evm_led_disable();
   return(OK);
}
/*******************************************************/
/* CodecExBlkCapPb – CODEC Example: Block Capture and   */
/*                   Playback                           */
/* This function provides an example of capturing a     */
/* block of audio data and playing it back for          */
/* verification                                         */
/*******************************************************/
static int CodecExBlkCapPb(int  micSel, int mode)
{
   c_srate  sRate;
   int   status;
   volatile u16 cnt = 2;
bufFlag[0] = 0;
bufFlag[1] = 0;
   if (micSel)
      sRate = VOICE_BAND_SAMPLE_RATE;
   else
      sRate = PRO_AUDIO_SAMPLE_RATE;
memset(toneBuf, '\0', 2*BUF_SIZE);
   // configure CODEC
   if (codec_init())
   {
      //printf("Error initializing codec");
      return(ERROR);
   }
   codec_sample_rate(sRate);
   if (micSel)
   {
      if (codec_input(MIC_SEL, MIC_SEL))
      {
         //printf("Error performing left and right channel
                  mic select\n");
         return(ERROR);
      }
      //printf("Begin speaking into the microphone when
                the LED illuminates. \n");
   }
   else
   {
      if (codec_input(LINE_SEL, LINE_SEL))
      {
         //printf("Error performing left and right channel
                  line select\n");
         return(ERROR);
      }
```

```
      printf("Play input source when LED illuminates.\n");
   }
   stop = False;
   if (codec_cont_capture(toneBuf[0], toneBuf[1],
      BUF_SIZE, 0) == ERROR)
   {
      printf("Error playing buffer to codec\n");
      return (ERROR);
   }
   if (codec_cont_play(toneBuf[0], toneBuf[1], BUF_SIZE,
       0) == ERROR)
   {
      printf("Error playing buffer to codec\n");
      return (ERROR);
   }
   while (cnt--)
      evm_delay_msec( 5000 );
   stop = True;
   MCBSP_RX_RESET(1);
   MCBSP_TX_RESET(1);
   evm_led_disable();
   return(ERROR);
}
/*******************************************************/
/* _serialRxIsr() – Copies received data directly to    */
/* transmit register                                    */
/*******************************************************/
interrupt void _serialRxIsr(void)
{
   *(u16 *)(DXR1_ADDR(1)) = *(u16 *)(DRR1_ADDR(1));
   *(u16 *)(DXR2_ADDR(1)) = *(u16 *)(DRR2_ADDR(1));
}
/*******************************************************/
/* genTone() – Generate tone samples based using the    */
/*             sinusoidal table and the variable sig.   */
/* pBuf – I/O buffer for generated data                 */
/* size – number of shorts in buffer                    */
/* ch –  LEFT, RIGHT, or BOTH                           */
/*******************************************************/
static void genTone(s16 *pBuf, u16 size, u16 ch)
{
   u16 i = 0;
   s16 temp;
   for (i=0; i<size; i++)
   {
      /* index = round–off, grab bits 7 through 14, and */
      /* shift right by 7                               */
      temp = sine_table[ ((sig.offset + 0x0040) &
            INTEGER_PART) >> 7 ];
      if ( (ch == LEFT) || (ch == BOTH) )
         pBuf[i] = temp;
      else
         pBuf[i] = 0;
```

```
         ++i;
         if ( (ch == RIGHT) || (ch == BOTH) )
            pBuf[i] = temp;
         else
            pBuf[i] = 0;
         /* calculate next offset. Q7 + Q7 */
         sig.offset += sig.delta;
      }
   }
```

Chapter 6

Daughterboard Memory Access Code Examples

This chapter provides code samples for several different types of daughterboard memory access.

Topic **Page**

6.1 Sequential 16-Bit Read Accesses 6-2

6.2 Sequential 32-Bit Read Accesses 6-2

6.3 Consecutive 32-Bit Odd/Even Read Accesses 6-3

6.4 Consecutive 32-Bit Write Accesses 6-3

6.1 Sequential 16-Bit Read Accesses

The C54x can access sequential 16-bit data memory locations on a daughterboard. Here is an example that reads 50 consecutive 16-bit data memory locations. This example could also be used for writes using "*dataPtr++ = buffer[i]" instead.

```
/*****************************************************/
/* Sequential 16-bit daughterboard read accesses      */
/*****************************************************/
// Ext. data accesses are in the range of 0x8000-0xFFFF
// Set data pointer to start of external memory range.
// NOTE: This 0x8000 appears as 0x0000 to the daughter-
// board. The DM_PG bits can be used to access multiple
// 32Kword pages.
dataPtr = (volatile unsigned int *)0x8000;
// For 16-bit accesses DB_WIDE must be 0 and DB_32ODD
// is not used. Only DM_SEL must be set along with the
// desired data memory page (DM_PG[4..0]).
CPLD_DMCTRL_REG = 0x84;     // access 5th 32K data page
// store fifty 16-bit words into buffer array
for (i=0; i<50; i++)
{
   // read 16-bit word from daughterboard memory
   buffer[i]   = *dataPtr++;
}
```

6.2 Sequential 32-Bit Read Accesses

The C54x can access sequential 32-bit data memory locations on a daughterboard. Here is an example that reads 50 consecutive 32-bit data memory locations.

```
/*****************************************************/
/* Sequential 32-bit daughterboard read accesses      */
/*****************************************************/
// Ext. data accesses are in the range of 0x8000-0xFFFF
// Set data pointer to start of external memory range.
dataPtr = (volatile unsigned int *)0x8000;
// store fifty 32-bit words into buffer array
for (i=0; i<50; i++)
{
   // control DB_32ODD bit based on source address
   CPLD_DMCTRL_REG = 0xC0 | ((i & 1) << 5);
   // read LSW first at source address
   buffer[i*2]  = *dataPtr++;
   // read MSW second at source address plus 1
   buffer[i*2+1] = *dataPtr;
   // NOTE: dataPtr is not incremented since it is
   //       the source address for the next access.
}
```

6.3 Consecutive 32-Bit Odd/Even Read Accesses

Consecutive 32-bit data memory access to just even or odd addresses can be done also if the DB_32ODD bit stays fixed. Here is an example of just reading the even addresses. For odd address accesses, the DB_32ODD bit should be set to 1 (CPLD_DMCTRL_REG = 0xE0).

```
/*******************************************************/
/* 32–bit daughterboard read accesses to even addresses */
/*******************************************************/
dataPtr = (volatile unsigned int *)0x8000;
// control DB_32ODD bit based on source address
// NOTE: DB_32ODD bit will stay fixed at 0 for even addr.
CPLD_DMCTRL_REG = 0xC0;
// store fifty even address 32–bit words into buffer array
for (i=0; i<50; i++)
{
   // read LSW first at source address
   buffer[i*2]  = *dataPtr++;
   // read MSW second at source address plus 1
   buffer[i*2+1] = *dataPtr++;
}
```

6.4 Consecutive 32-Bit Write Accesses

The C54x can also write to sequential 32-bit data memory locations on a daughterboard. Here is an example that writes 50 consecutive 32-bit data memory locations.

```
/*******************************************************/
/* Sequential 32–bit daughterboard write accesses       */
/*******************************************************/
// Ext. data accesses are in the range of 0x8000–0xFFFF
// Set data pointer to start of external memory range.
dataPtr = (volatile unsigned int *)0x8000;
// write fifty 32-bit words from buffer array to DB mem.
for (i=0; i<50; i++)
{
   // control DB_32ODD bit based on source address
   CPLD_DMCTRL_REG = 0xC0 | ((i & 1) << 5);
   // write MSW first at destination address plus 1
   *(dataPtr+1) = buffer[i*2];
   // write LSW second at destination address
   *dataPtr++ = buffer[i*2+1];
}
```