TMS320C62x Multichannel Evaluation Module Reference Guide

Literature Number: SPRU308
February 1999

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

Preface

Read This First

About This Manual

This manual provides technical reference information for the TMS320C62x (’C62x) multichannel evaluation module (McEVM). It includes support software documentation, application programming interface (API) references, and hardware descriptions for the ’C62x McEVM.

The ’C62x McEVM is a peripheral component interconnect (PCI) plug-in card that is compliant with the *PCI Local Bus Specification Revision 2.1*. The ’C62x McEVM helps you evaluate characteristics of the ’C62x digital signal processor (DSP) to determine if it meets your application requirements. It is a high-performance platform targeted for multichannel telecom and datacom applications. The ’C62x McEVM is intended for use in a PCI expansion slot inside the PC™ . It can also be operated outside the PC on a desktop or lab bench with the use of an external power supply and emulator (XDS510 or XDS510WS). The power supply and emulator are not included in the kit.

’C62x McEVM support software and APIs enable you to use the board to create applications for the ’C62x. Software utilities are provided with the ’C62x McEVM for board diagnostics, board configuration, and common object file format (COFF) DSP application loading. McEVM schematics and logic equations are included in this manual to ease your hardware development efforts and reduce time to market.

This manual assumes that you are familiar with working in a Windows 95™ , Windows 98™ , or a Windows NT™ environment and understand general and technical PC terminology. This manual specifically addresses the ’C62x McEVM and its support software. Detailed information about the ’C62x DSP and TI code development support tools is provided separately (see the *Related Documentation From Texas Instruments* section in this *Preface* for a list of documents and ordering information). For up-to-date information on the ’C62x McEVM, as well as related products, visit the ’C6000 website at http://www.ti.com/sc/c6000/.

How to Use This Manual

This reference guide provides the following types of information about the ’C62x McEVM:

- Chapter 1 describes the theory of operation for the ’C62x EVM hardware, including key component identification, detailed information about each functional area, and interface descriptions.
- Chapter 2 describes the host support software utilities. It includes a complete API reference that allows you to write your own applications for the ’C62x McEVM and also includes example applications that use the API calls.
- Chapter 3 describes the DSP support software. It includes an overview of all the DSP support software components and a complete API reference that allows you to write your own DSP applications for the ’C62x McEVM. Examples applications that use the API calls are also provided.

❑ **Reference material**, consisting of Appendixes A through E, provides quick reference information for the ’C62x McEVM:

- Appendix A provides the McEVM connector pinouts.
- Appendix B contains the McEVM schematics.
- Appendix C provides the McEVM complex programmable logic device (CPLD) equations.
- Appendix D summarizes the McEVM PCI configuration EEPROM.
- Appendix E provides a glossary of terms used in this manual.

Notational Conventions

This document uses the following conventions:

❑ Program listings, program examples, and interactive displays are shown in a `special typeface`. Some examples use a **`bold version`** for emphasis; interactive displays use a **`bold version`** 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 with the evm6x_close() function highlighted for emphasis:

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;
    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        exit(–1);
    }
. . .
    evm6x_close( h_board );
```

- ❑ In syntax descriptions, the instruction or command is in a **bold face**, and parameters are in *italics*. 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 command syntax:

  evm6xldr *filename*

- ❑ Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of a command that has optional parameters.

  evm6xtst [*options*] [*log_filename*]

  evm6xtst is the command. This command has two optional parameters, indicated by *options* and *log_filename*.

- ❑ Device pins often are represented in groups. Device pin group notation consists of the pin name followed by brackets containing the range of pins included in the group. A colon separates the numbers in the range. For example, GD[31:0] represents the global data bus pins on a device.

- The TMS320C62x family of devices is referred to as the ’C62x. The following abbreviations are used in this manual for TI devices on the ’C62x McEVM:

Abbreviation	Device Definition
’C6201	TMS320C6201
’ALVCH16244	SN74ALVCH16244
’CBT3257	SN74CBT3257
’CBTD3384	SN74CBTD3384
’LVT125	SN74LVT125
’ALVCH16245	SN74LVCH16245

Information About Cautions and Warnings

This book contains cautions and warnings.

CAUTION

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

WARNING

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to you.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation From Texas Instruments

The following books describe the ’C62x processor and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800)477–8924. When ordering, please identify the book by its title and literature number.

TMS320C6000 Assembly Language Tools User’s Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the ’C6000 generation of devices.

TMS320C6x C Source Debugger User’s Guide (literature number SPRU188) tells you how to invoke the ’C6x simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

TMS320C6000 Optimizing C Compiler User’s Guide (literature number SPRU187) describes the ’C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the ’C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the ’C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, multichannel buffered serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer’s Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

TMS320C6000 Technical Brief (literature number SPRU197) gives an introduction to the ’C6000 platform of digital signal processors, development tools, and third-party support.

XDS51x Emulator Installation Guide (literature number SPNU070) describes the installation of the XDS510™, XDS510PP™, and XDS510WS™ emulator controllers. The installation of the XDS511™ emulator is also described.

TMS320 DSP Development Support Reference Guide (literature number SPRU011) describes the TMS320 family of digital signal processors and the tools that support these devices. Included are code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

TMS320C6x Peripheral Support Library Programmer's Reference (literature number SPRU273) describes the contents of the 'C6x peripheral support library of functions and macros. It lists functions and macros both by header file and alphabetically, provides a complete description of each, and gives code examples to show how they are used.

TMS320C62x Multichannel Evaluation Module User's Guide (literature number SPRU285) is a high-performance, multichannel telephony platform for the development, analysis, and testing of 'C62x digital signal processor (DSP) algorithms and applications. The 'C62x McEVM allows you to evaluate the 'C62x DSP and algorithms to determine if they meet your application requirements.

TMS320C6201, TMS320C6201B Digital Signal Processors Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 and TMS320C6201B fixed-point DSPs and provides pinouts, electrical specifications, and timings for the devices.

Related Documentation

For up-to-date information on the 'C62x McEVM, as well as related products, visit the 'C6000 website at:

http://www.ti.com/sc/docs/dsps/tools/c6000/index.htm

You can use the following specification to supplement this reference guide:

PCI Local Bus Specification Revision 2.1, PCI Special Interest Group, June 1, 1995.

http://www.pcisig.com/specs.html

MVIP-90 Reference Manual, GO-MVIP, Inc. 1990 (http://www.mvip.org/MemOrder.htm)

The MVIP Book, GO-MVIP, Inc., ISBN 0-936648-76-7.

Trademarks

320 Hotline On-line, VelociTI, XDS510, XDS510PP, XDS510WS, and XDS511 are trademarks of Texas Instruments Incorporated.

ABEL and Synario are registered trademarks of the DATA I/O Corporation.

Altera, ByteBlaster, and MAX+ PLUS are trademarks of the Altera Corporation.

AMCC is a registered trademark of Applied Micro Circuits Corporation.

IBM and PC are trademarks of International Business Machines Corporation.

Intel and Pentium are trademarks of Intel Corporation.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

MVIP, MVIP-90, MVIP Bus and Multi-Vendor Integration Protocol are trademarks of GO-MVIP, Inc.

OpenWindows, Solaris, and SunOS are trademarks of Sun Microsystems, Inc.

SPARCstation is a trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

X Window System is a trademark of the Massachusetts Institute of Technology.

Obtaining Technical Support

Before contacting Texas Instruments Technical Support, please have the following information available:

❑ Assembly number of your ’C62x McEVM (D600860-0001) located on the bottom side of the board

❑ Revision number of your ’C62x McEVM located in parentheses next to the assembly number on the bottom side of the board

❑ Serial number located on the bottom side of the board

❑ Record of the ’C62x McEVM confidence test utility results that identifies potential problems and other revision numbers (software, EEPROM, CPLD). *TMS320C62x McEVM User’s Guide,* Chapter 3, *Running the Board Confidence Test*, explains how to run the test.

❑ Computer’s PCI BIOS brand name and version number

- ❑ Amount of memory in your computer system
- ❑ Version of the software and operating environment you are using such as Windows NT 4.0
- ❑ Version of the code generation tools you are using
- ❑ Version of the debugger you are using
- ❑ If you are using Windows 95, print out a report of your system configuration by performing the following steps:

  1) Right click on the My Computer icon on the desktop.
  2) Select the Properties menu item.
  3) Select the Device Manager tag.
  4) Select the Print button.
  5) Select the System summary radio button.
  6) Click on the OK button to print a system resource summary.

- ❑ If you are using Windows NT, perform the following steps to get system information:

  1) Select the Run... menu item from the Windows NT Start menu.
  2) Type **winmsd** at the Open prompt, and press Enter.
  3) Select the Resources tab of the Windows NT Diagnostics window.
  4) Click on the Print button to get a report.
  5) Click on OK at the Create Report window.

  Have this system resource summary available when you contact technical support.

  Note:

  Check the system resource summary to see if the IRQ assigned to TI TMS320C62x McEVM is shared with another device. If it is, this is probably the problem.

Once you have this information ready, contact Texas Instruments Technical Support as specified in the *If You Need Assistance* section that follows.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580	
TI Literature Response Center U.S.A.	(800) 477-8924	
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285	
U.S. Technical Training Organization	(972) 644-5580	
DSP Hotline		Email: dsph@ti.com

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32
Email: epic@ti.com		
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

TMS320C62x McEVM Hardware

This chapter describes the TMS320C62x McEVM hardware, its key components and how they operate, and its various interfaces. Detailed programmer interface information such as memory maps, register definitions, interrupt usage, and required software initialization tasks are also provided.

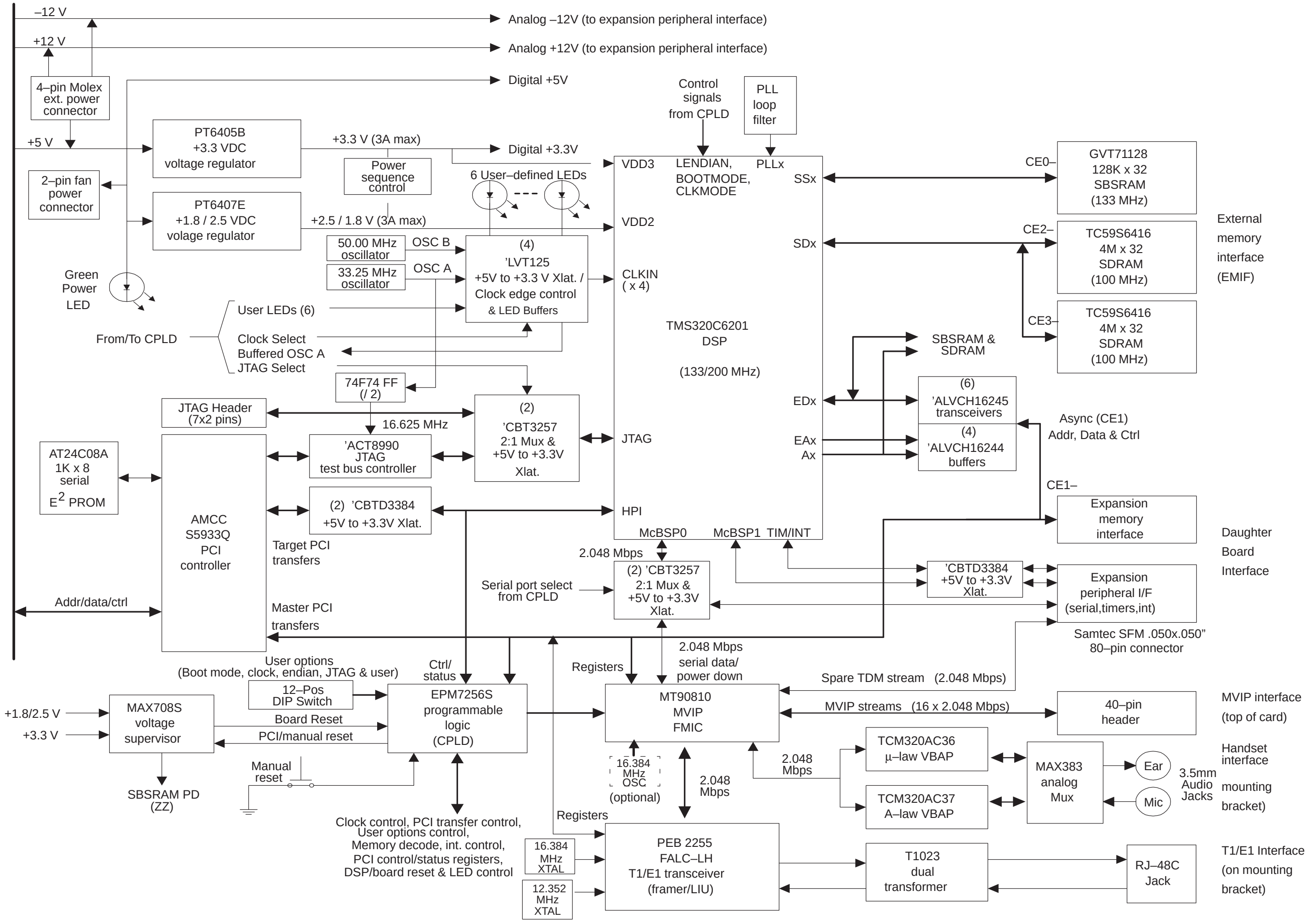
The hardware can be divided into 14 functional areas. Each of these areas is discussed in detail in this chapter.

Topic **Page**

1.1 TMS320C62x McEVM Hardware Detailed Block Diagram

Figure 1–1 provides a detailed block diagram of the ’C62x McEVM hardware. It identifies the key components described in this chapter and shows how they interface to each other.

Figure 1–1. TMS320C62x McEVM Detailed Block Diagram


–12 V
Analog –12V (to expansion peripheral interface)
+12 V
Analog +12V (to expansion peripheral interface)
Digital +5V
4–pin Molex ext. power connector
+5 V
PT6405B +3.3 VDC voltage regulator
+3.3 V (3A max)
Digital +3.3V
Power sequence control
2–pin fan power connector
PT6407E +1.8 / 2.5 VDC volage regulator
+2.5 / 1.8 V (3A max)
6 User–defined LEDs
Control signals from CPLD
PLL loop filter
VDD3
LENDIAN, BOOTMODE, CLKMODE
PLLx
SSx
VDD2
SDx
50.00 MHz oscillator
OSC B
33.25 MHz oscillator
OSC A
(4) 'LVT125 +5V to +3.3 V Xlat. / Clock edge control & LED Buffers
CLKIN (x 4)
Green Power LED
User LEDs (6)
From/To CPLD
Clock Select
Buffered OSC A
JTAG Select
TMS320C6201 DSP
(133/200 MHz)
74F74 FF (/ 2)
JTAG Header (7x2 pins)
16.625 MHz
(2) 'CBT3257 2:1 Mux & +5V to +3.3V Xlat.
'ACT8990 JTAG test bus controller
JTAG
AT24C08A 1K x 8 serial E2 PROM
AMCC S5933Q PCI controller
(2) 'CBTD3384 +5V to +3.3V Xlat.
HPI
Target PCI transfers
Master PCI transfers
Addr/data/ctrl
McBSP0
McBSP1
TIM/INT
2.048 Mbps
Serial port select from CPLD
(2) 'CBT3257 2:1 Mux & +5V to +3.3V Xlat.
EDx
EAx
Ax
CE0–
GVT71128 128K x 32 SBSRAM (133 MHz)
CE2–
TC59S6416 4M x 32 SDRAM (100 MHz)
CE3–
TC59S6416 4M x 32 SDRAM (100 MHz)
External memory interface (EMIF)
SBSRAM & SDRAM
(6) 'ALVCH16245 transceivers
(4) 'ALVCH16244 buffers
Async (CE1) Addr, Data & Ctrl
CE1–
Expansion memory interface
Daughter Board Interface
'CBTD3384 +5V to +3.3V Xlat.
Expansion peripheral I/F (serial,timers,int)
Samtec SFM .050x.050” 80–pin connector
User options (Boot mode, clock, endian, JTAG & user)
Ctrl/ status
Registers
2.048 Mbps serial data/ power down
Spare TDM stream (2.048 Mbps)
+1.8/2.5 V
+3.3 V
MAX708S voltage supervisor
12–Pos DIP Switch
Board Reset
PCI/manual reset
EPM7256S programmable logic (CPLD)
MT90810 MVIP FMIC
MVIP streams (16 x 2.048 Mbps)
40–pin header
MVIP interface (top of card)
SBSRAM PD (ZZ)
Manual reset
16.384 MHz OSC (optional)
2.048 Mbps
2.048 Mbps
TCM320AC36 µ–law VBAP
TCM320AC37 A–law VBAP
MAX383 analog Mux
Ear
Mic
3.5mm Audio Jacks
Handset interface mounting bracket)
Clock control, PCI transfer control, User options control, Memory decode, int. control, PCI control/status registers, DSP/board reset & LED control
Registers
16.384 MHz XTAL
12.352 MHz XTAL
PEB 2255 FALC–LH T1/E1 transceiver (framer/LIU)
T1023 dual transformer
RJ–48C Jack
T1/E1 Interface (on mounting bracket)

1.2 TMS320C6201 DSP

The TMS320C6201 DSP is the brain of the ’C62x McEVM. The ’C6201 DSP has the following key features:

- VelociTI™ advanced very long instruction word (VLIW) architecture
  - Load/store architecture
  - Instruction packing for reduced code size
  - 100% conditional instructions for faster execution
  - Intuitive, reduced instruction set computing with RISC-like instruction set
- CPU
  - Eight independent functional units (including two 16-bit multipliers with 32-bit results and six arithmetic logic units (ALUs) with 32/40 bit results)
  - 32 32-bit registers
  - 1600 million instructions per second (MIPS)
  - 5-ns cycle time
  - Up to eight 32-bit instructions per cycle
  - Byte-addressable with 8-, 16- and 32-bit data
  - 32-bit address range
  - Little- and big-endian support
  - Saturation
  - Normalization
  - Bit-field instructions (extract, clear, left-most bit detection)
- Memory/peripherals
  - Glueless external memory interface to synchronous memories such as SBSRAM and SDRAM
  - Glueless external memory interface to asynchronous memories such as SRAM and EPROM
  - 4-channel direct memory access (DMA)
  - Host port interface (HPI) with dedicated auxiliary DMA channel providing access to entire processor memory space
  - Two multichannel buffered serial ports (McBSPs) for direct interfacing to telecommunication, audio, and other serial devices

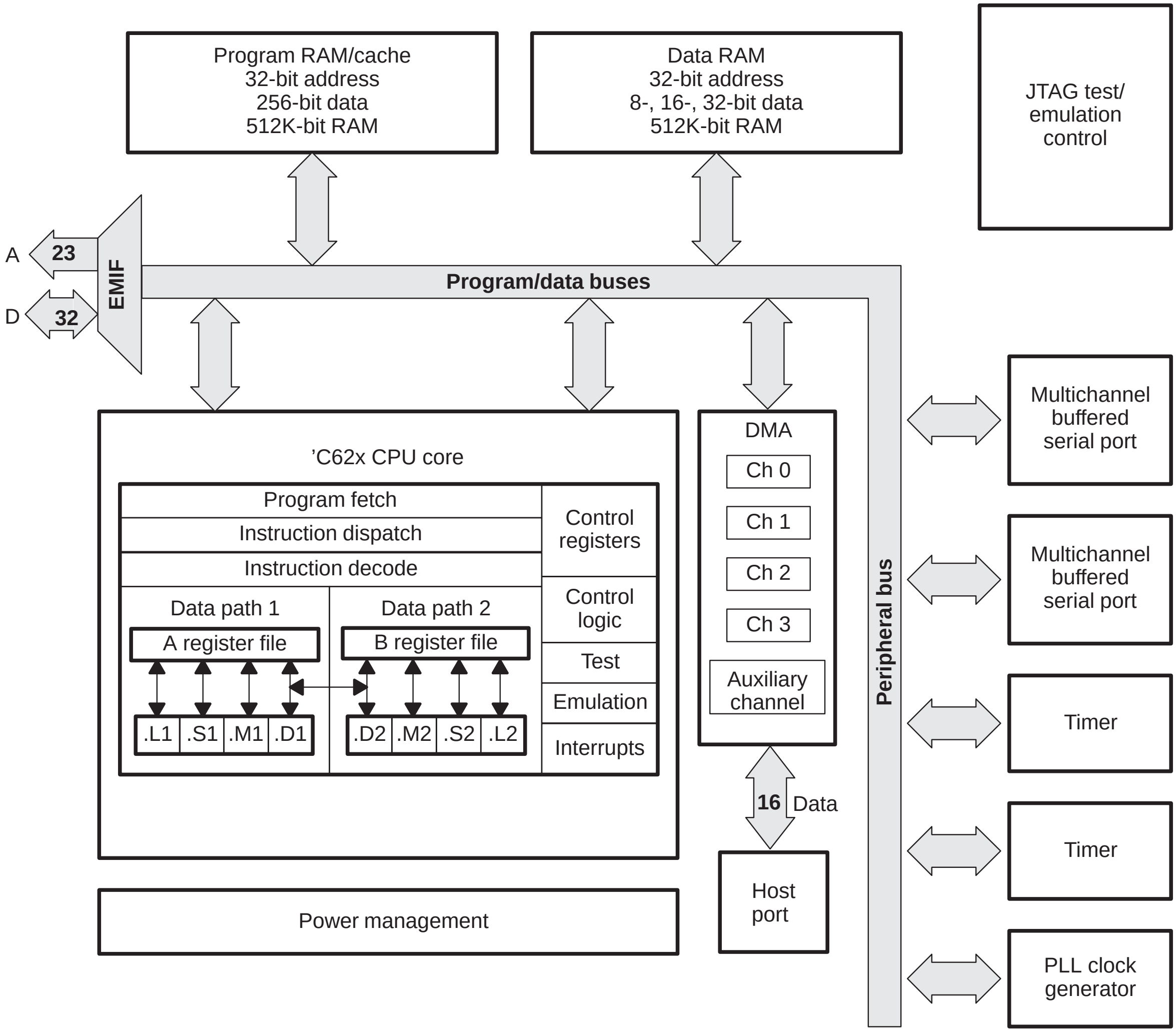
  - Two general-purpose timers
  - Multiply-by-4 phase locked loop (PLL) and multiply-by-1 PLL-bypass options
  - 1M-bit on-chip memory (2K × 256 bits of program memory/64K bytes of data memory)

- ❑ Miscellaneous
  - IEEE-1149.1 (JTAG) boundary-scan compatible for emulation and test support
  - 352-lead ball grid array (BGA) package
  - 0.18-micron/5-level metal process with CMOS technology
  - 3.3-V I/O and 1.8-V internal core voltages

Figure 1–2 shows the ’C6201 core, peripherals, and external interfaces.

Because the ’C62x McEVM is a hardware reference design intended to provide you with maximum flexibility, it supports all of the DSP’s external interfaces. The ’C62x McEVM uses the PLL clock generator interface to support multiply-by-1 and multiply-by-4 clock modes with CPU clock rates of 33.25 MHz, 50 MHz, 133 MHz, and 200 MHz. The JTAG test/emulation interface supports both embedded and external emulation for source code debugging. The control interface resets the DSP, provides external interrupts, chooses data endian mode, and selects the DSP’s boot method. The external memory interface (EMIF) supports synchronous SBSRAM and SDRAM memories and asynchronous accesses to the CPLD registers, Multi-Vendor Integration Protocol (MVIP) flexible MVIP interface circuit (FMIC), T1/E1 transceiver, PCI controller, and expansion memory. The EMIF is also brought out to an expansion memory interface connector for daughterboard use. The host port interface (HPI) is used for bidirectional data transfers between the PC and the DSP. The timer interfaces are provided on the expansion peripheral interface connector for daughterboard use. The multichannel buffered serial ports provide interfaces to an onboard MVIP FMIC and/or a daughterboard connected to the expansion peripheral interface.

Figure 1–2. TMS320C6201 Core, Peripherals, and External Interfaces



1.3 DSP Clocks

The ’C62x McEVM provides quad DSP clock support, which allows you to run the DSP core at these rates:

❑ Full speed (200 MHz)
❑ Speed of the SBSRAM (133 MHz)
❑ Multiply-by-1 clock modes (33.25 MHz and 50 MHz)

Typically, you will use either the full-speed or SBSRAM-speed DSP clock. Depending on the application, one speed may be more efficient than the other. You can perform benchmarks using both clocks to determine optimal performance for a particular application.

The McEVM uses two half-size oscillators, along with a TI SN74LVT125 quad bus buffer device, to provide 33.25- and 50-MHz CLKIN signals to the ’C6201. The use of the ’LVT125 device provides multiple functions, including 5-V to 3.3-V signal level translation, clock selection, and clock edge control with fast rise and fall times required by the DSP.

The clock selection is made via a DIP switch during external operation or via a software switch for internal PCI operation. Clock select control logic in the complex programmable logic device (CPLD) controls the ’LVT125 buffer output enable that corresponds to the selected clock oscillator. The CPLD provides break-before-make clock switching between the two clock oscillators to prevent any output contention.

The McEVM provides user options for selecting the CLKIN frequency (33.25 or 50-MHz) and the clock mode (multiply-by-1 or multiply-by-4), so the CLKOUT1 (CPU clock) frequency can be 33.25 MHz, 50 MHz, 133 MHz, or 200 MHz (see Table 1–1). The SBSRAM clock (SSCLK) can be configured by the DSP software to be one-half the CPU clock (CLKOUT1) or the same as CLKOUT1. The SDRAM clock (SDCLK) is always one-half of CLKOUT1. The CPLD also controls the DSP’s CLKMODE and PLLFREQ control inputs based on the clock mode selection.

Table 1–1. DSP Clock Summary

Clock Source	Clock Mode	CLKOUT1	SSCLK (1/2 rate/1× rate)	SDCLK
33.25 MHz	Multiply-by-1	33.25 MHz	16.625/33.25 MHz	16.625 MHz
50.00 MHz	Multiply-by-1	50.00 MHz	25/50 MHz	25 MHz
33.25 MHz	Multiply-by-4	133 MHz	66.5/133 MHz	66.5 MHz
50.00 MHz	Multiply-by-4	200 MHz	100 MHz†	100 MHz

† 1× SSCLK rate is invalid when CLKOUT1 is 200 MHz.

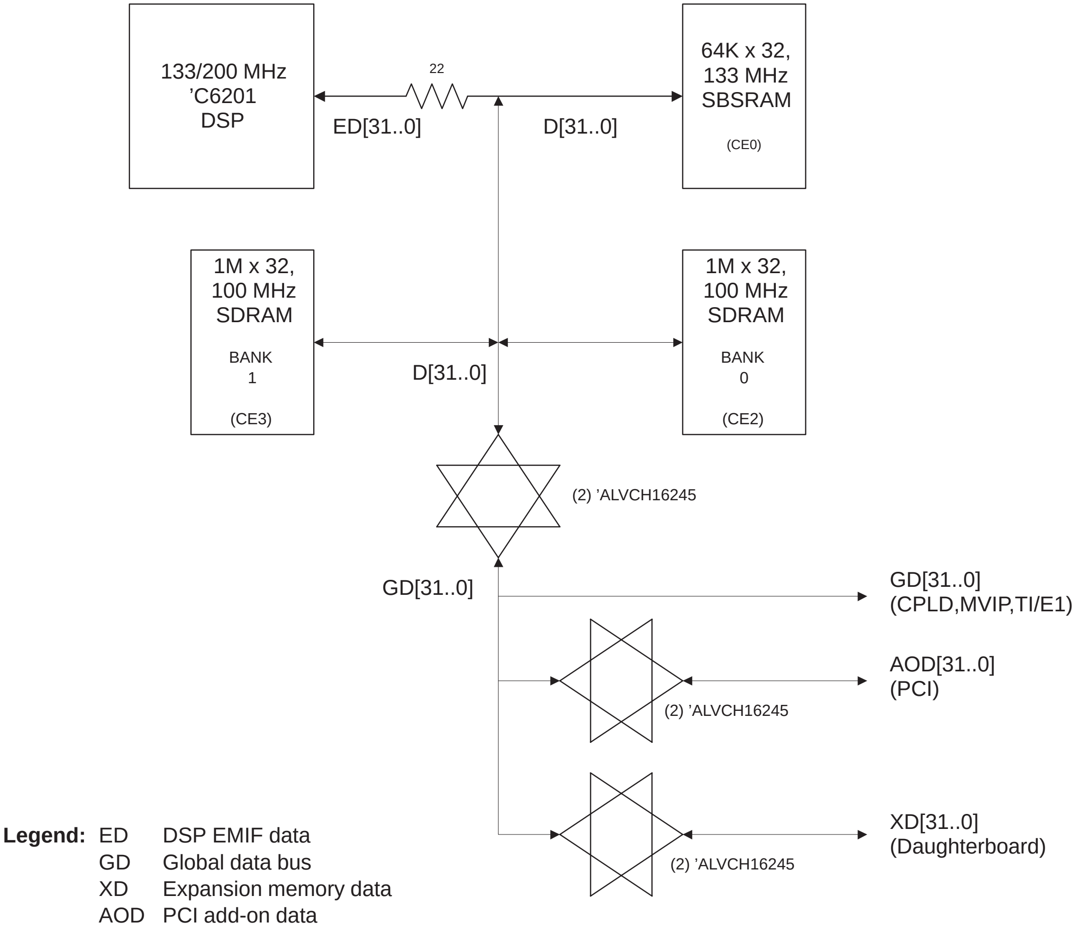
1.4 External Memory

The ’C62x McEVM provides one bank of 128K × 32-bit words of 7.5-ns (133-MHz) SBSRAM and two banks of 4M × 32-bit words of 10-ns (100-MHz) SDRAM. An expansion memory connector is also provided to enable asynchronous memory and memory-mapped devices to be added using a daughterboard. The external SBSRAM and SDRAM devices on the board are 3.3-V devices. The expansion memory connector is able to support both 3.3-V and 5-V devices because ’ALVCH16245 5-V-tolerant buffers are used.

The EMIF address and data busses are connected to the high-speed SBSRAM and SDRAM memories using series termination. Buffers and transceivers are used to buffer signals to other memory-mapped devices to preserve signal integrity, limit loading on the ’C6201 outputs, and provide the necessary drive to meet timing requirements. Low-voltage data transceivers that are 5-V tolerant provide voltage translation and the necessary drive to a daughterboard.

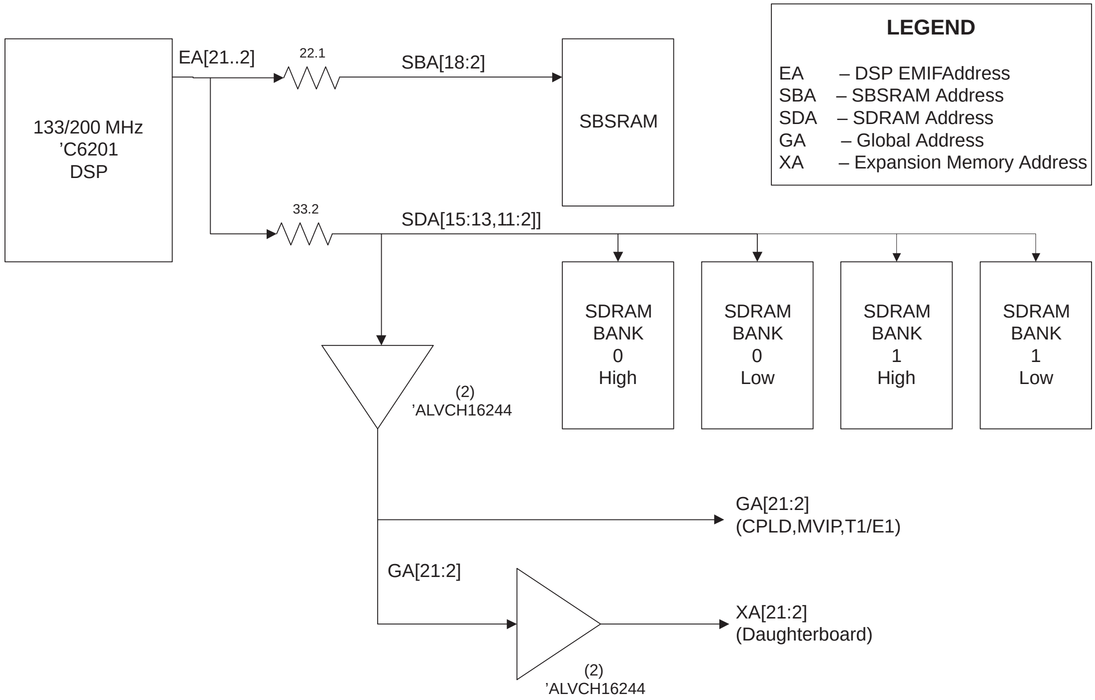
Figure 1–3 and Figure 1–4 show the topologies of the EMIF data and address busses on the McEVM, respectively.

Figure 1–3. EMIF Data Bus Topology



Note: The transceiver signal assignments provide a functional overview only. The actual signal assignments are optimized for layout.

Figure 1–4. EMIF Address Bus Topology



Note: The buffer signal assignments provide a functional overview only. The actual signal assignments are optimized for layout.

1.4.1 SBSRAM

The McEVM provides 128K × 32-bit words of SBSRAM. The SBSRAM used on the McEVM is directly connected to the DSP with no glue logic or buffers required, because the ’C6201 EMIF provides a direct interface to industry-standard SBSRAM devices. The bank of SBSRAM is mapped into the DSP’s CE0 memory space, allowing it to be used for program booting, assuming that it has been initialized by host software or the emulator. It has a 7.5-ns cycle time (133 MHz) and provides 128K × 32-bit words of memory in a single 100-pin thin quad flat pack (TQFP) package.

The SBSRAM device can be clocked by the ’C6201 at the CPU clock speed when operating at 133 MHz, one-half the clock speed when operating at 200 MHz, or at either the same as or one-half the clock speed when operating in the multiply-by-1 clock mode (33.25 MHz and 50 MHz). This selection is made based on the setting of the SSCRT bit in the EMIF global control register. Additionally, various boot modes inherently select the clock speed used for the

SBSRAM interface. You must correctly select the SBSRAM clock rate for the selected CPU clock rate.

The ’C6201 EMIF provides signals that directly correspond to the SBSRAM pins. Because the ’C6201 can generate a new address every cycle, the SBSRAM’s $\overline{\text{ADV}}$ control input, which allows the device to generate its own addresses internally, does not have to be used. This SBSRAM control input is pulled high on the McEVM.

The SBSRAM device supports a snooze power-down mode when its ZZ input signal is a logic high. The ZZ input is controlled directly by the voltage supervisor’s logic high reset output; so, when the board is held in reset, the SBSRAM is disabled and placed into a power-down mode. Data in the SBSRAM is retained during the power-down mode.

1.4.2 SDRAM

The McEVM provides two banks of 4M × 32-bit words of SDRAM. Each bank is comprised of two 1M × 4 banks × 16-bit devices. The 3.3-V SDRAM used on the ’C62x McEVM requires no glue logic to interface to the DSP because the ’C6201 EMIF generates all the required SDRAM control and refresh signal sequences. The two banks of SDRAM are mapped into the DSP’s CE2 and CE3 memory spaces. Each bank uses all of its respective memory space’s 16M-byte address space.

The SDRAM devices are always clocked at one-half the CPU clock rate. This means that when the DSP core runs at 133 MHz, the SDRAM runs at 66.5 MHz (15 ns), and when the DSP core runs at 200 MHz, the SDRAM runs at 100 MHz (10 ns). The McEVM’s SDRAM devices are rated for 100-MHz operation.

The four SDRAM devices are driven by the DSP’s SDCLK output signal that is optimally routed in an ’H’ configuration. Data, address, and control signals to the SDRAM devices are directly connected (via series termination resistors) to the DSP using carefully designed routing. There is no margin in the DSP’s SDRAM timing for any buffering with bus operation of 100 MHz.

The SDRAM must be refreshed periodically to maintain its data. The ’C62x McEVM uses the DSP’s SDRAM refresh capability so the RFEN bit in the DSP’s SDRAM control register must be set to 1. The refresh period can be a maximum of 15.625 µs.

The McEVM’s CPLD includes SDRAM enable bits that are controlled via the DSP memory-mapped SDCNTL register. After reset is released, these SDRAM enables, which are connected to the SDRAM clock enable (CKE) pins, default to active, enabling the SDRAMs. When the board is held in reset or DSP software explicitly clears the register control bits, the SDRAMs are disabled and placed into a power-down mode.

An additional use of the CPLD SDRAM enable bits is the ability to independently disable the CE2 and CE3 SDRAM banks to allow these memory spaces to be used for asynchronous expansion memory on a daughterboard. When an SDRAM bank is disabled, the respective memory space can be used for asynchronous expansion memory.

1.4.3 Expansion Memory

The McEVM provides an asynchronous expansion memory interface connector (J8) to allow you to add memory or memory-mapped devices via a daughterboard. The expansion memory interface is mapped into the lower 3M bytes of the DSP’s 4M-byte asynchronous CE1 memory space. Expansion memory in the CE1 space is addressed from 0x1000000–12FFFFF in MAP 0 and 0x1400000–16FFFFF in MAP 1 mode. The upper 1M bytes of the CE1 memory space is allocated for onboard peripherals. This division of the CE1 memory space allows both the onboard devices and the expansion memory interface to coexist without conflicts.

Because the McEVM’s MVIP FMIC, T1/E1 transceiver, PCI controller, and CPLD registers are also accessed in the DSP’s asynchronous CE1 memory space, the CPLD provides transceiver control logic that prevents the expansion memory space from conflicting with the onboard use of the CE1 space. The CPLD monitors the CE1 signal along with the upper address signals (EA[21:20]) to determine when the lower 3M-byte expansion memory space is being accessed and enables the expansion memory transceivers accordingly. CE1 decoding in the upper 1M byte is handled by the CPLD for control of onboard peripherals.

The EMIF CE2 and CE3 memory space enables are available on the expansion peripheral interface connector (J9). These two memory spaces can also be used for asynchronous memory on the daughterboard when their respective SDRAM enable bits are not asserted in the CPLD register. The SDRAM enable bits control the SDRAM clock enables, as well as enabling the expansion memory transceivers to be turned on during CE2 and CE3 memory space accesses. This capability supports applications that do not require one or both banks of SDRAM, but need to interface to faster or additional asynchronous memory on a daughterboard.

All expansion memory interface signals are buffered using TI low-voltage, 5-V tolerant buffers/transceivers to allow both 3.3- and 5-V devices to be used on the daughterboard and to isolate the daughterboard from the onboard EMIF. The three memory space enables (CE1–CE3) are buffered versions of the DSP outputs and are not generated by decode logic. This allows fast daughterboard logic to be used as required for the application without incurring additional delay. The expansion memory transceivers isolate the daughterboard and onboard data busses to prevent bus contention.

One potential use of the expansion memory interface is to provide nonvolatile memory such as ROM or Flash memory on a daughterboard that can be used for ROM boot operation. This allows the McEVM to be autobooted at power up and reset with an application stored in nonvolatile memory. The McEVM can operate in this mode inside the PC or, more typically, in an external operating environment.

1.4.4 DSP Memory Maps

Table 1–2 and Table 1–3 show the ’C62x McEVM DSP memory maps for MAP 0 and MAP 1 modes, respectively.

Table 1–2. TMS320C62x McEVM DSP Memory Map (MAP 0)

Start Address	End Address	External Memory Space	Size (Bytes)	Description
00000000	0007FFFF	CE0	512K	SBSRAM
00080000	00FFFFFF	CE0	16M – 512K	Unused
01000000	012FFFFF	CE1	3M	Asynchronous expansion memory
01300000	0130003F	CE1	64	PCI add-on registers
01300040	0130FFFF	CE1	64K – 64	Unused
01310000	01310003	CE1	4	PCI FIFO register
01310004	0133FFFF	CE1	192K – 4	Unused
01340000	0134000F	CE1	16	MVIP FMIC
01340010	0134FFFF	CE1	64K – 16	Unused
01350000	013500FF	CE1	256	T1/E1 transceiver
01350100	0137FFFF	CE1	192K – 256	Unused
01380000	01380037	CE1	56	DSP control/status registers
01380038	013FFFFF	CE1	512K – 56	Unused
01400000	0140FFFF	N/A	64K	Internal program memory (IPM)
01410000	017FFFFF	N/A	4M – 64K	Reserved (future IPM)
01800000	01BFFFFF	N/A	4M	Internal peripherals
01C00000	01FFFFFF	N/A	4M	Reserved
02000000	02FFFFFF	CE2	16M	SDRAM (bank 0) or asynchronous expansion memory
03000000	03FFFFFF	CE3	16M	SDRAM (bank 1) or asynchronous expansion memory
04000000	7FFFFFFF	N/A	1984M	Reserved
80000000	8000FFFF	N/A	64K	Internal data memory (IDM)
80010000	803FFFFF	N/A	4M–64K	Reserved (future IDM)
80400000	FFFFFFFF	N/A	2044M	Reserved

Table 1–3. TMS320C62x McEVM DSP Memory Map (MAP 1)

Start Address	End Address	External Memory Space	Size (Bytes)	Description
00000000	0000FFFF	N/A	64K	Internal program memory (IPM)
00010000	003FFFFF	N/A	4M–64K	Reserved (future IPM)
00400000	007FFFFF	CE0	256K	SBSRAM
00800000	013FFFFF	CE0	16M – 256K	Unused
01400000	016FFFFF	CE1	3M	Asynchronous expansion memory
01700000	0170003F	CE1	64	PCI add-on registers
01700040	0170FFFF	CE1	64K – 64	Unused
01710000	01710003	CE1	4	PCI FIFO register
01710004	0173FFFF	CE1	192K – 4	Unused
01740000	0174000F	CE1	16	MVIP FMIC
			64K – 16	Unused
01750000	017500FF	CE1	256	T1/E1 transceiver
01750100	0177FFFF	CE1	192K – 256	Unused
01780000	01780037	CE1	56	DSP control/status registers
01780038	017FFFFF	CE1	512K – 56	Unused
01800000	01BFFFFF	N/A	4M	Internal peripherals
01C00000	01FFFFFF	N/A	4M	Reserved
02000000	02FFFFFF	CE2	16M	SDRAM (bank 0) or optional asynchronous expansion memory
03000000	03FFFFFF	CE3	16M	SDRAM (bank 1) or optional asynchronous expansion memory
04000000	7FFFFFFF	N/A	1984M	Reserved
80000000	8000FFFF	N/A	64K	Internal data memory (IDM)
80010000	803FFFFF	N/A	4M–64K	Reserved (future IDM)
80400000	FFFFFFFF	N/A	2044M	Reserved

1.4.5 DSP EMIF Registers

The DSP EMIF registers must be initialized before the external memory on the ’C62x McEVM can be accessed by DSP or host software. The DSP EMIF registers must be initialized by the DSP software before it accesses external memory in the no-boot and ROM-boot modes. When the HPI-boot mode is selected, the host software must initialize the EMIF registers via the HPI before HPI memory accesses are performed on external memory. This section identifies the required and recommended EMIF register values for proper McEVM operation.

The EMIF global control register must be initialized to enable the various DSP output clocks and select clock polarities, the SBSRAM clock rate (one-half the CPU clock or the same as the CPU clock), and the requester arbitration mode. The CLKOUT2 signal is routed to the expansion peripheral connector for daughterboard use. The ’C62x McEVM does not use the CLKOUT1 output, so these clocks should be disabled to minimize EMI emissions. The SDCLK output is used to clock the McEVM’s four SDRAM devices. The SBSRAM clock rate (SSCLK) can be either one-half the CPU clock or the same as the CPU clock when the CPU clock rate is 33.25 MHz, 50 MHz, or 133 MHz. When the CPU clock rate is 200 MHz, only one-half the CPU clock rate is valid for SSCLK. The requester arbitration mode selection is application dependent. For a one-half-rate SSCLK operation, a value of 0x3068 is recommended. For a full-rate SSCLK operation, a value of 0x306C is recommended.

The EMIF CE space control registers for the CE0–CE3 must be initialized to select the external memory configuration of the ’C62x McEVM. *The CE1 memory space control register must be configured for a strobe period of 3 CLKOUT1 cycles for proper operation.* Table 1–4 summarizes the CE space allocation of the ’C62x McEVM.

Table 1–4. TMS320C62x McEVM CE Memory Space Initialization Summary

CE Memory Space	Memory Type	Memory Characteristics	Control Register Address	Control Register Value	
CE0	SBSRAM	133 MHz maximum	0x1800008	0x40	
CE1	PCI controller, CPLD registers, MVIP FMIC, T1/E1 transceiver, and expansion memory	32-bit async, strobe = 3	0x1800004	200 MHz: 133 MHz: 50 MHz: 33.25 MHz:	0x50F50323 0x40F40323 0x10D10321 0x10D10321
CE2	SDRAM (bank 0)	100 MHz maximum	0x1800010	0x30	
CE3	SDRAM (bank1)	100 MHz maximum	0x1800014	0x30	

Because the ’C62x McEVM includes SDRAM, the EMIF SDRAM control and timing registers must be initialized to select timing and device width, enable refresh, initialize the SDRAM devices, and control the SDRAM refresh period. The SDRAM timing selection is dependent on the CPU clock speed, so the host or DSP software must determine its clock speed to properly initialize the timing control bits. The host and DSP can each read CPLD registers to determine the CPU clock speed based on the clock selection (33.25 MHz or 50 MHz) and clock mode (multiply-by-1 or multiply-by-4). The SDRAM devices have a t_{RC} (refresh/active-to-refresh/active) period of 84 ns, a t_{RP} (precharge-to-active) period of 24 ns, and a t_{RCD} (active-to-read/write) period of 24 ns. Based on the determined CPU clock period, the TRC, TRP, and TRCD fields of the EMIF SDRAM control register can be initialized based on the CLKOUT2 period, as summarized in Table 1–5.

Table 1–5. EMIF SDRAM Control Register Timing Fields

CPU Clock (MHz)	CLKOUT2 Period (ns)	TRC Field	TRP Field	TRCD Field
33.25	60	1	0	0
50	40	2	0	0
133	15	5	1	1
200	10	8	2	2

The SDRAM control register’s INIT bit must be set to 1 to initialize the SDRAM in each CE space configured for SDRAM (CE2/CE3). The RFEN bit must be set to 1 to enable EMIF SDRAM refreshes. The SDWID bit must also be set to 1 to select two, 16-bit SDRAM devices per SDRAM CE space.

The SDRAM timing register, which selects the SDRAM refresh period, must be initialized for a maximum period of 15.625 µs. The refresh period is based on CLKOUT2 cycles, so the SDRAM timing register period must have maximum values as summarized in Table 1–6.

Table 1–6. EMIF SDRAM Timing Register Values

CPU Clock (MHz)	CLKOUT2 Period (ns)	Maximum Register Value
33.25	60	0x103
50	40	0x185
133	15	0x410
200	10	0x619

See the *TMS320C6201/C6701 Peripherals Reference Guide* for detailed information about the DSP’s EMIF registers.

1.5 Expansion Interfaces

The ’C62x McEVM provides two expansion connectors that allow a daughterboard to be connected to the board. Daughterboards can be used to extend the capabilities of the McEVM and to provide custom and application-specific I/O. One expansion connector provides the DSP’s asynchronous EMIF, and the other provides access to the DSP’s peripherals and control/status signals. Both connectors also provide power to the daughterboard.

Most of the expansion connector signals are buffered so that the daughterboard cannot directly influence the operation of the McEVM board. The use of low-voltage, 5-V tolerant interface devices allows the use of either 5- or 3.3-V devices to be used on the daughterboard.

The ’C62x McEVM’s expansion memory and peripheral interfaces are provided with two dual-row, 80-pin connectors. These surface-mount connectors are low profile and have a 0.050-inch (1.27-mm) pitch. The recommended mating connectors provide 0.465-inch board spacing, allowing ample space for daughterboard components.

The expansion memory interface connector has a reference designator of J8 on the McEVM. The expansion peripheral interface connector is J9. See Appendix A for the pinouts of the expansion connectors.

1.5.1 Expansion Memory Interface

The expansion memory interface provides the DSP’s asynchronous EMIF signals to a daughterboard. External asynchronous memories and memory-mapped devices can be added to the McEVM, including nonvolatile memory that can be used to boot the McEVM upon reset.

The expansion memory interface includes:

❑ **20 external address signals (EA[21:2]).** All of the DSP’s 20 external address signals are available on the expansion memory interface, allowing up to 4M bytes of external memory to be addressed. However, because the CE1 memory space must be shared with onboard peripherals, only the lower 3M bytes are available to a daughterboard. If CE2 or CE3 is used for external asynchronous memory instead of SDRAM, an additional 4M bytes in each memory space can be addressed.

❑ **32 external data signals (ED[31:0]).** All of the DSP’s 32 external data signals are available on the expansion memory interface to support full 32-bit word accesses to the daughterboard.

❑ **CE1 memory space enable.** The DSP’s CE1 memory space enable is available on the expansion memory interface to allow asynchronous accesses to daughterboard memory and memory-mapped devices.

- **Four byte enables (BE[3:0]).** The DSP’s four byte enables are available on the expansion memory interface to support byte (8-bit), halfword (16-bit), and word (32-bit) daughterboard memory accesses.
- **Four asynchronous control signals.** The DSP’s asynchronous control signals ($\overline{\text{ARE}}$, $\overline{\text{AWE}}$, $\overline{\text{AOE}}$, and ARDY) are provided to control asynchronous memory accesses to a daughterboard.
- **Power signals**. The expansion memory interface also provides ground, 5-V, and 3.3-V power signals to the daughterboard.

1.5.2 Expansion Peripheral Interface

The expansion peripheral interface provides the DSP’s peripheral signals to a daughterboard. This peripheral expansion capability allows serial devices and communication devices to be added to the McEVM via a daughterboard.

The expansion peripheral interface includes:

- **Seven signals for each of the serial ports (McBSP0 and McBSP1).** The DSP’s seven McBSP1 signals are available on the expansion peripheral interface. These signals are buffered by a ’CBTD3384 device to support both 5- and 3.3-V serial devices using McBSP1 on a daughterboard.

  The DSP’s seven McBSP0 signals are also available when the DSP software controls onboard ’CBT3257 multiplexers that connect them to the expansion connector rather than the MVIP FMIC. This architecture provides a daughterboard with access to both of the DSP’s serial ports, which is useful in many DSP applications. Because a ’CBT3257 multiplexer is used, both 5- and 3.3-V serial devices can use McBSP0 on a daughterboard.
- **Two input/output signals for each of the timers (timer 0 and timer 1).** The expansion peripheral interface includes each of the DSP timers’ input and output signals. This allows timer signals to be sent to the daughterboard, or timer input or events to be counted to come from the daughterboard. Each timer has one input and one output signal.
- **Interrupt, interrupt acknowledge, and identification signals.** A DSP external interrupt (DB_INT) is included on the expansion peripheral interface to allow the daughterboard to interrupt the ’C6201 to notify it of data transfers and other significant events. This interrupt is pulled down on the McEVM, so the daughterboard must drive it high to interrupt the DSP. Additionally, the DSP’s interrupt acknowledge (IACK) and interrupt identification number signals (INUM[3:0])are available to the daughterboard.
- **Four DMA completion flags.** The DMA action complete flags (DMAC[3:0]) are available to the daughterboard on the expansion periph-

eral interface. These pins provide a method of feedback to external logic generating an event for each DMA channel. The DMAC pins can also be used for general-purpose output control signals controlled from the DSP's DMA channel secondary control register.

❑ **Four general-purpose input/output flags.** Two general-purpose control inputs and two status outputs are brought to the expansion peripheral interface to allow the DSP to control and monitor various signals on a daughterboard. The XCNTL[0:1] and XSTAT[0:1] signals can be controlled with DSP software by accessing the CPLD's DSP memory-mapped CNTL and STAT registers, respectively.

❑ **Power-down signal.** The DSP's power-down indication signal (DSP_PD) is also brought to the expansion peripheral interface so that a daughterboard can be powered down, if desired.

❑ **Reset signal.** The expansion peripheral interface also provides a reset signal that is active low when the board is in the reset state. This allows circuitry on the daughterboard to be set in a known state. The reset signal is asserted for a minimum of 140 ms upon power up, via a manual reset pushbutton or under software control. A memory-mapped register bit in the CPLD's CNTL register allows DSP software to directly control this reset signal.

❑ **CLKOUT2 signal for the synchronization clock.** The DSP's CLKOUT2 signal (CPU clock divide by 2) is brought out to the peripheral expansion interface for synchronization needs on daughterboards.

❑ **Buffered CE2 and CE3 signals for possible memory space use when the respective SDRAMs are disabled.** The DSP's CE2 and CE3 memory space decodes are buffered and brought out to the expansion peripheral interface to provide additional fast memory decodes. This can be useful on daughterboards that have multiple devices that need fast memory decodes. The CE2 and CE3 are dedicated to SDRAM use on the McEVM board, but the EMIF control registers can be initialized for asynchronous operation, which disables the respective SDRAM banks and allows expansion asynchronous memory to be used instead. The CE2 and CE3 SDRAM enable bits in the CPLD's DSP memory-mapped registers must also be used to shut down the respective SDRAM bank and allow the CPLD logic to enable the external data transceivers for the CE2/CE3 accesses.

❑ **Power signals.** The expansion peripheral interface also provides ground, 12-V, –12-V, 5-V, and 3.3-V power signals to the daughterboard.

1.5.3 Daughterboard

The ’C62x McEVM supports the mating of a daughterboard that has two 80-pin 0.050-inch × 0.050-inch TFM-series connectors from Samtec. The recommended mating connector (part number TFM–140–32–S–D–LC) is a surface-mount connector that provides a 0.465-inch mated height.

The McEVM supports two sizes of daughterboards that both use the two 80-pin connectors. The small daughterboard measures approximately 3.15 inches × 3.4 inches and mounts in the center of the board over the low-profile buffers and memories. This format is intended for daughterboards that do not require an I/O connection on the mounting bracket.

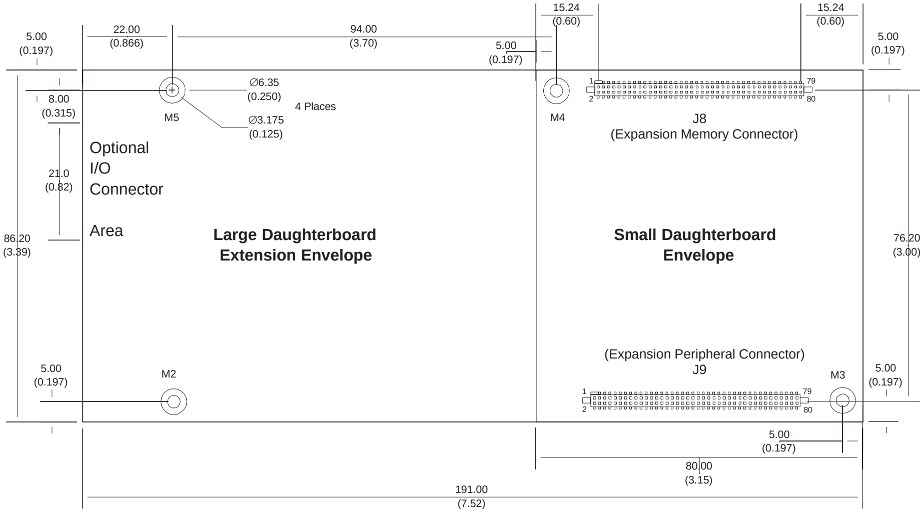
The large daughterboard measures approximately 7.5 inches × 3.4 inches and mounts from the center of the board over to the mating connector end of the board. This format is intended for daughterboards that require an I/O connection on the mounting bracket or need more space than the small daughterboard provides.

A daughterboard mounts with its component side down. This ensures that the PCI height requirement is met and no components are exposed to possible damage due to board insertions and extractions.

The McEVM provides four standoff mounting holes to support daughterboard connections. Mounting holes M3 and M4 support small daughterboards, and all four mounting holes support large daughterboards. The four mounting holes are electrically connected to the digital ground plane to provide additional daughterboard grounding.

Figure 1–5 shows the small and large daughterboard envelopes, the relationship between the two expansion connectors, and the relative location of the four mounting holes on the component side of the ’C62x McEVM board.

Figure 1–5. Daughterboard Envelopes and Connections on the ’62x McEVM



CAUTION

It is important to note that this figure is showing the top side of the McEVM, not the actual daughterboard.

Notes:

1) All dimensions are shown in millimeters. Inch dimensions are shown in parentheses.
2) Drawing shows daughterboard envelopes and connections on the component side of McEVM board.
3) Standard-size daughterboard is 80.0 mm × 86.2 mm (3.15 in. × 3.39 in.).
4) Full-size daughterboard is 191.0 mm × 86.2 mm (7.52 in. × 3.39 in.).
5) Daughterboard connectors are Samtec .050-in. × .050-in. Micro Strips (SFM–140–L2–S–D–LC).
6) Daughterboard mating connectors are Samtec TFM–140–32–S–D–LC.
7) Mating height is 0.465 in. (11.81 mm).
8) There are four plated holes (M2–M5) on the McEVM for standoff mounting.
9) Mounting holes M2–M5 are electrically connected to digital ground.

1.6 PCI Interface

The ’C62x McEVM’s peripheral component interconnect (PCI) interface provides plug-and-play functionality along with the ability to support high-speed target (slave) and initiator (master) modes of data transfers.

The plug-and-play feature of PCI is intended to eliminate the resource conflicts associated with ISA cards that result from user configuration of addresses and interrupts. PCI devices each provide a configuration register space within the system that can be accessed by the host prior to it being mapped into the system memory or I/O space. Access to the configuration registers is the key to PCI’s plug-and-play functionality. The PC’s BIOS executes configuration cycles after reset to identify devices on the PCI bus and to determine each of their system resource requirements, such as I/O and memory space and interrupts. PCI devices are automatically configured by the PC BIOS, to prevent system resource conflicts. The McEVM’s Windows drivers obtain information from the McEVM’s PCI controller’s configuration registers to determine where the board is located and what interrupt it uses. This allows you to simply plug the board into a PCI slot without setting any jumpers or switches.

The PCI bus operates synchronously at up to 33 MHz with a multiplexed 32-bit address/data bus. The power of PCI is its support for multiple devices to master the bus and communicate in bursts at up to 132M bytes/second (33 MHz × 4 bytes/word). A burst consists of a single 32-bit address phase, followed by sequential 32-bit data words. Only one device can be mastering the bus at any one time, but for the period that it does, it can burst data at that rate if its hardware can keep up. If it is not fast enough, a ready signal is used to throttle the transfer at the rate at which it can read or write data. Because there are typically multiple devices on the PCI bus, such as the video controller, they must all timeshare the bandwidth, so the effective transfer rate for each device is typically much lower than 132M bytes/second. The key to maximizing transfer throughput on the PCI bus is to use burst transfers when possible to avoid repetitive PCI bus arbitration and the overhead associated with single-word transfers. The PCI bridge provides a hidden central arbitration mechanism where multiple bus masters can request and be granted the bus. It also controls the length of the bursts that each device can generate. PCI transfer rates are very machine-dependent because burst transfer support varies among the various bridges used in different PCs.

1.6.1 PCI Interface Implementation

The ’C62x McEVM implements a fully-compliant PCI Revision 2.1 interface using an industry-standard application-specific integrated circuit (ASIC) (AMCC™ part number S5933). The S5933 PCI controller interfaces directly to the PCI bus connector (P1) and handles all of the PCI-side transactions, freeing the McEVM hardware from having to handle them directly. The ’C62x McEVM’s 32-bit PCI interface operates at up to 33 MHz in a 5-V signaling environment. The McEVM cannot be used in a 3.3-V PCI slot.

The S5933 provides PCI configuration registers that are always accessible to host software—even before the board has been mapped into the system resources. Accesses to these configuration registers are made by host software by specifying the device’s bus number, device number, and function number. The PCI configuration registers are in a unique address space, so they are not directly mapped into either the host’s I/O or memory spaces. Nonvolatile, serial EEPROM memory on the ’C62x McEVM is used to store PCI configuration information for the board including its vendor and device IDs, memory space requirements, and operational parameters. The McEVM’s vendor ID is 0x104C (Texas Instruments), and its device ID is 0x1003.

The S5933 provides a glueless interface to the serial EEPROM that can be accessed from both the host and DSP software via memory-mapped register access. The contents of the EEPROM are automatically loaded into S5933 configuration registers at power-up reset to identify the system resource requirements and operating characteristics of the McEVM. This information is used by the PC BIOS for system resource allocation.

The McEVM uses an AT24C08A 1Kx8 serial EEPROM. The S5933 only requires 64 bytes for configuration information. Another 64 bytes is reserved for future use, so there are 896 free bytes, located from offsets 0x80 to 0x3FF, that can be used to store miscellaneous information, if desired.

The S5933 provides five base address registers (BARs). Each BAR is initialized upon power up by the nonvolatile memory to the desired size of a memory-mapped region for the device. The BIOS overwrites the BAR values with the address that it allocates to each region. The ’C62x McEVM takes advantage of all five BARs. The first BAR (BAR0) is reserved for the PCI controller’s operation registers. The other four BARs (BAR1–BAR4) are used to interface to the JTAG test bus controller (TBC), McEVM control and status registers, and the ’C6201 HPI. Table 1–7 summarizes the ’C62x McEVM PCI BAR definitions.

Table 1–7. TMS320C62x McEVM PCI BAR Definitions

BAR Number	Size (Bytes/DWORDs)	Bus Width/Bits Used	Description
0	64/16	32/32	S5933 PCI operation registers
1	128/32	32/16	JTAG TBC registers
2	128/32	32/8	McEVM board control/status registers
3	16/4	16/32	HPI control, address, and data registers
4	256K/64K	16/32	HPI data register (with autoincrement)

The S5933 provides three physical interfaces:

❑ PCI bus
❑ Add-on bus
❑ A nonvolatile memory interface

Data movement can occur between the PCI bus and the add-on bus, as well as between the add-on or PCI bus and the nonvolatile memory. Data transfers between the PCI and add-on buses can take place through mailbox registers, FIFOs, or the pass-through data path, which is a generic memory-mapped interface.

Mailbox registers are used to pass single 32-bit values between the host and DSP. Interrupts can be used to indicate when the mailbox registers are full and empty. The S5933 has eight mailboxes that are useful for passing command and status information between the host and the DSP. There are four incoming (host-to-DSP) and four outgoing (DSP-to-host) mailboxes. The host incoming and the DSP outgoing mailboxes are the same internally. The DSP incoming and the host outgoing mailboxes are the same internally. The mailbox status can be monitored from both sides in two ways. A mailbox status register available to both sides indicates the empty/full status of bytes within the mailboxes. The mailboxes can also be configured to generate interrupts to the host and DSP. One outgoing and one incoming mailbox on each side can be configured to generate interrupts.

FIFO transfers between the PCI and add-on buses can be performed under software control or directly by hardware using the device as a bus master. This means that the DSP software can access the S5933 FIFOs directly to read and write data from and to the PCI bus, or it can program its DMA controller to handle the transfers automatically in the background.

The pass-through data path is used when the McEVM is a PCI target for JTAG TBC and DSP HPI transactions.

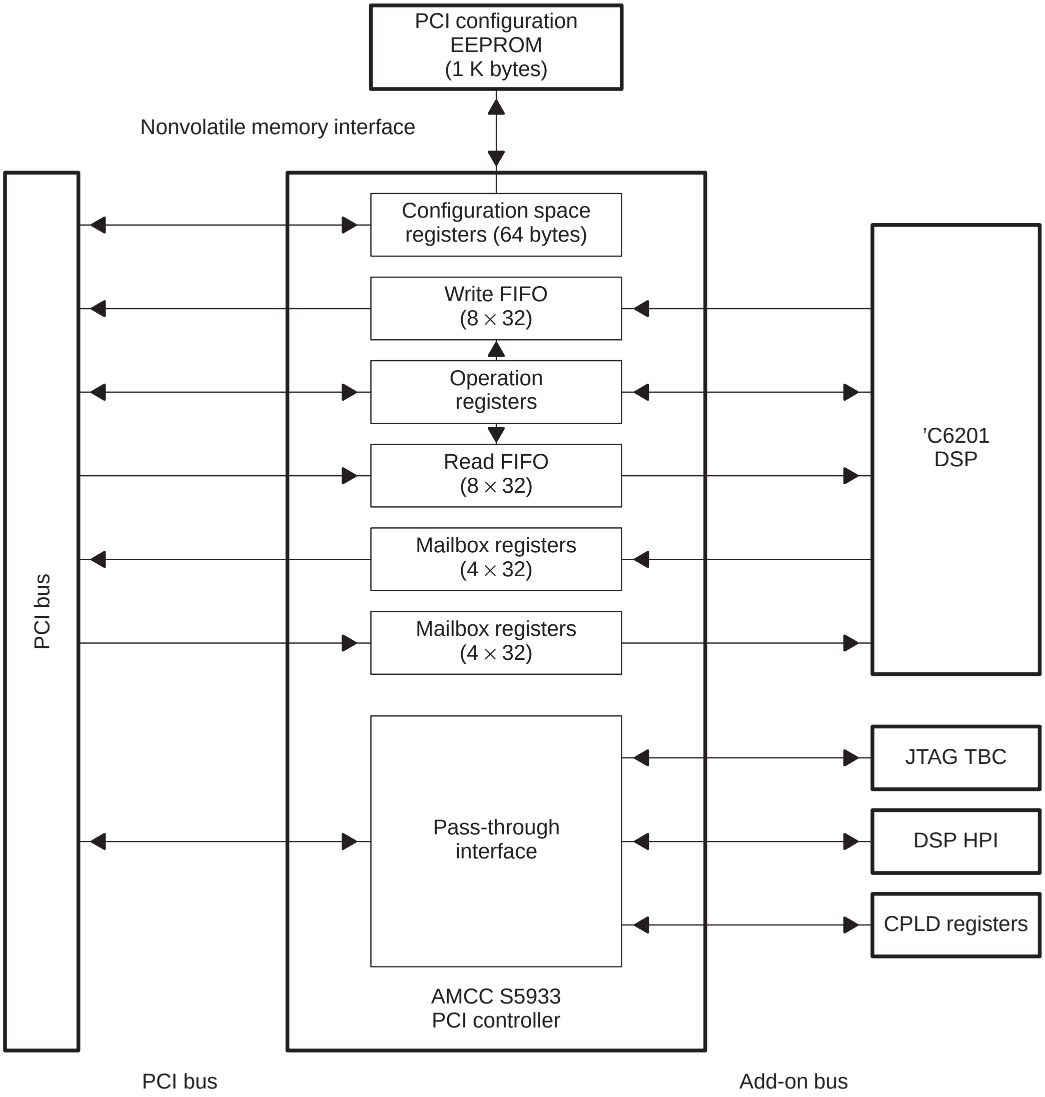
The S5933 registers and FIFOs are memory mapped into both the host and DSP memory spaces. The host can read the S5933 configuration registers to determine where the McEVM is located in physical memory and what host interrupt has been assigned to it. The host software also configures the S5933 through the registers for master bus transfers because it knows the physical addresses of its memory buffers and other PCI devices.

The McEVM is configured for PCI-initiated bus master transfers. In this configuration, the DSP cannot access the master read/write address and transfer count registers, and the DSP cannot be interrupted when the transfer counts reach 0. The DSP can access the other S5933 registers to read/write mailboxes, determine interrupt status, and read/write the FIFOs. Because the DSP typically uses DMA to control bus master transfers, a ’C62x DMA interrupt can be used to indicate the end of a transfer.

The S5933 supports both PCI (INTA#) and add-on interface (IRQ#) interrupts. This allows the host software and DSP software to be notified upon PCI events, such as the end of bus master transfers (host only), mailbox empty/full, and bus errors. The DSP is interrupted via its EXT_INT4 interrupt when the S5933 asserts its IRQ interrupt. The device also allows the host to control the system reset (SYSRST#) signal directly from software, so a software board reset can be invoked via the PCI bus.

Figure 1–6 provides a block diagram that shows the ’C62x McEVM’s PCI interface implementation based on the S5933 PCI controller.

Figure 1–6. PCI Interface Implementation



1.6.2 PCI Controller Operation Registers

The S5933 PCI operation registers allow the host software to:

- ❑ Configure the device and monitor its status
- ❑ Read and write mailboxes
- ❑ Reset the board
- ❑ Initiate bus master transfers
- ❑ Access the FIFOs

Table 1–8 identifies the S5933 PCI operation registers, along with their BAR0 offsets and access types. All registers are 32 bits wide and are addressed on doubleword (DWORD) boundaries.

Table 1–8. S5933 PCI Bus Operation Registers

Host Byte Address	PCI Operation Register	Description	Access
BAR0 + 0x00	OMB1	Outgoing mailbox register 1	Read/write
BAR0 + 0x04	OMB2	Outgoing mailbox register 2	Read/write
BAR0 + 0x08	OMB3	Outgoing mailbox register 3	Read/write
BAR0 + 0x0C	OMB4	Outgoing mailbox register 4	Read/write
BAR0 + 0x10	IMB1	Incoming mailbox register 1	Read only
BAR0 + 0x14	IMB2	Incoming mailbox register 2	Read only
BAR0 + 0x18	IMB3	Incoming mailbox register 3	Read only
BAR0 + 0x1C	IMB4	Incoming mailbox register 4	Read only
BAR0 + 0x20	FIFO	FIFO register port (bidirectional)	Read/write
BAR0 + 0x24	MWAR	Master write address register	Read/write
BAR0 + 0x28	MWTC	Master write transfer count register	Read/write
BAR0 + 0x2C	MRAR	Master read address register	Read/write
BAR0 + 0x30	MRTC	Master read transfer count register	Read/write
BAR0 + 0x34	MBEF	Mailbox empty/full status register	Read only
BAR0 + 0x38	INTCSR	Interrupt control/status register	Read/write
BAR0 + 0x3C	MCSR	Bus master control/status register	Read/write

1.6.3 PCI Add-On Bus Operation Registers

The S5933 add-on bus operation registers allow the ’C6201 DSP software to:

❑ Read and write mailbox messages

❑ Read and write FIFO data

❑ Control interrupts

❑ Provide read and write access to the PCI controller’s configuration EEPROM

The PCI add-on bus operation registers are mapped into the DSP’s CE1 asynchronous memory space starting at 0x01300000 (MAP 0) or 0x01700000 (MAP 1).

Table 1–9 summarizes the S5933 PCI controller’s 64-byte bank of add-on bus operation registers. Registers in the address offset range of 0x24–0x33 are not available because they are either under direct hardware control or are inaccessible with PCI-initiated bus mastering.

Table 1–9. PCI Add-on Bus Operation Registers Summary

DSP Byte Address MAP 1 (MAP 0)	Add-on Bus Operation Register	Description	Access
01700000 (01300000)	AIMB1	Add-on incoming mailbox register 1	Read only
01700004 (01300004)	AIMB2	Add-on incoming mailbox register 2	Read only
01700008 (01300008)	AIMB3	Add-on incoming mailbox register 3	Read only
0170000C (0130000C)	AIMB4	Add-on incoming mailbox register 4	Read only
01700010 (01300010)	AOMB1	Add-on outgoing mailbox register 1	Read/write
01700014 (01300014)	AOMB2	Add-on outgoing mailbox register 2	Read/write
01700018 (01300018)	AOMB3	Add-on outgoing mailbox register 3	Read/write
0170001C (0130001C)	AOMB4	Add-on outgoing mailbox register 4	Read/write
01700020 (01300020)	AFIFO	Add-on FIFO register port	Read/write
01700024–01700033 (01300024–01300033)	–	Unavailable	None
01700034 (01300034)	AMBEF	Add-on mailbox empty/full status	Read only
01700038 (01300038)	AINT	Add-on interrupt control	Read/write
0170003C (0130003C)	AGCSTS	Add-on general control/status	Read/write

The PCI add-on FIFOs can be accessed at offset 0x20 of the PCI add-on register address space (AFIFO) for general-purpose data transfers. However, the dedicated FIFO access addresses (0x01710000/0x01310000) must be used during PCI bus master transfers.

1.6.4 PCI Slave Support

The ’C62x McEVM’s PCI slave support enables the host, or other PCI device, to access its PCI controller, JTAG TBC, CPLD, and DSP HPI registers. Once the base addresses of these devices are obtained by the host driver upon initialization, their registers can be accessed like system memory. When the host accesses the McEVM’s slave devices, the S5933 PCI controller activates the pass-through bus to indicate that a data transfer is to take place. A state machine controller implemented in the McEVM’s CPLD manages the interface between the S5933 and the three targets. This state machine monitors and asserts signals that result in the transfer of data between the S5933’s pass-through data register and the target interfaces. The S5933 decouples the state machine from the PCI bus by handling all the PCI-side transactions, so it only has to manage the reading and writing of data between the S5933 and the targets via the pass-through interface.

1.6.4.1 JTAG TBC

The ’C62x McEVM’s onboard JTAG TBC enables host software to control the ’C6201 JTAG interface for testing and emulation purposes. The debugger uses this interface to control and monitor the DSP. The ’C62x McEVM debugger shipped with the McEVM kit can be used for source code debugging on the board without requiring any additional hardware, such as an XDS510.

The JTAG TBC has 24 registers that are memory mapped at DWORD offsets starting at the address defined by the S5933’s base address register 1 (BAR1). The TBC is a 16-bit device, so the upper 16 bits of data transfers are not used.

1.6.4.2 CPLD Registers

The ’C62x McEVM’s CPLD provides 11 memory-mapped board control and status registers that the host software accesses via the PCI bus. The registers are located at DWORD offsets starting at the address defined by the S5933’s BAR2. These registers allow the host software to control and monitor the ’C62x McEVM board. Host software can reset the TBC and the DSP, configure and poll interrupts, monitor several board status signals, control software switches, observe DIP switch and DSP option signals, check the revision number of the CPLD, and utilize two hardware semaphores. The CPLD registers are only eight bits wide, so the upper 24 bits of data transfers are not used.

See section 1.8.6, *PCI Memory-Mapped Board Control/Status Registers*, for details on the PCI memory-mapped CPLD registers.

Similar to the other target interfaces, the add-on bus state machine provides the control signals that enable data transfers between the PCI controller and the board control and status registers. Whenever the S5933 indicates either a PCI read or write access to the board control and status registers (BAR2), the state machine acknowledges it by asserting S5933 and register control signals required to complete the data transfer. The lower eight bits of the S5933 add-on data bus are connected to the registers. The CPLD latches the register address during the PCI address phase and the state machine asserts register clock enable and output enable signals. The register address decoding and enable signals are used to clock data into and read data from the nine registers.

No voltage translation is required because the CPLD has 5-V tolerant inputs.

1.6.4.3 DSP HPI Interface

The ’C62x McEVM provides DSP host port interface (HPI) access from the PCI bus, giving host software read and write access to all of the DSP’s memory space. The ’C62x McEVM supports both random and sequential accesses using the PCI controller’s BAR3 and BAR4 memory regions, respectively.

The three ’C6201 HPI registers are memory mapped on the ’C62x McEVM at DWORD offsets starting at the address defined by the S5933’s base address register 3 (BAR3). The fourth address in the BAR3 region simply aliases to the HPID register. The three registers map directly to the DSP’s HPI control (HPIC), address (HPIA), and data (HPID) registers. Table 1–10 summarizes the HPI registers mapped into BAR3.

Table 1–10. DSP HPI Registers (BAR3)

Host Byte Address	HPI Register	Description	Access
BAR3 + 0x00	HPIC	HPI control register	Read/write
BAR3 + 0x04	HPIA	HPI address register	Read/write
BAR3 + 0x08	HPID	HPI data register	Read/write
BAR3 + 0x0C	HPID	HPI data register (alias)	Read/write

Before HPI data transfers are performed successfully, the HPIC register must be properly initialized. The ’C62x McEVM handles HPI data transfers with the first halfword being the least significant word for compatibility with the little-endian PCI bus. The ’C6201 HPIC register bit HWOB must, therefore, be set to 1 to select this data ordering. A value of 0x00010001 should be written to the HPIC register at BAR3 offset 0. Any subsequent writes to the HPIC register, such as controlling host and DSP interrupts, must keep bits 0 and 16 high in the HPIC to maintain the low-word/high-word transfer order.

The ’C62x McEVM CPLD accepts data transfer requests from the S5933 PCI controller and asserts the DSP’s HPI control signals required to transfer two 16-bit words. The CPLD also handles the HPI ready control signal monitoring, so software handshaking is not required.

Byte enables asserted during the host write to the HPID register are transferred to the HPI byte enables ($\overline{\text{HBE}}$[1:0]) by the ’C62x McEVM’s CPLD. The ’C62x McEVM therefore supports byte, word, and doubleword data writes to anywhere in the DSP’s memory space. This capability is useful for modifying individual bytes and words in memory or memory-mapped registers without corrupting the other bytes in a 32-bit word. A standard 32-bit access to the HPID register results in all four bytes being modified in the DSP memory space. From a software perspective, the pointer to the HPID register is simply modified and cast accordingly to perform byte and word write operations. Assuming a doubleword pointer to the HPID register is named HPI_DATA, a write access to byte 3 (MSbyte) of a 32-bit word would be performed as follows in C:

```
*((*unsigned char)(((unsigned long)HPI_DATA)+3)) = ByteValue;
```

Host access to the HPI via the BAR3 memory region therefore allows random read and write accesses anywhere in the DSP memory space with byte, word, and doubleword support. BAR3 accesses require that the HPIA register be updated for each transfer to a different address.

The ’C62x McEVM also provides support, using the BAR4 memory region, for sequential data transfers between the host and the DSP by taking advantage of the HPI support for read and write autoincrement accesses. This capability allows blocks of data to be moved more efficiently between the host and the DSP without incurring the overhead associated with passing the memory address each time.

The HPI data registers in BAR4 are used when data is transferred between the host and DSP in autoincrement mode. This separate region eases the decoding and makes the transfers more efficient over the PCI bus with support for sequential and burst transfers. The HPI controller in the CPLD does not care about the addresses in this range, only that an access is within the address range. All accesses to BAR4 are written directly to the HPID register, assuming that the HPIA and HPIC registers have been initialized previously with autoincrement mode selected.

Sequential HPI data accesses with burst support is provided with host accesses to the memory-mapped region defined by BAR4. BAR4 provides a 64K DWORD (256K byte) memory window that host software can address as a linear array or a circular buffer. Accesses to the BAR4 memory region result in sequential data words being transferred to the HPI data register. The BAR4 offset address is ignored, since it is assumed that the HPI's address autoincrement feature has been selected. If the host cannot access memory past the 256K byte BAR4 allocation, the PCI controller ignores the request. The 'C62x McEVM's sequential address burst support eliminates the need for the host to arbitrate for the PCI bus as often and eliminates the need to pass the board for every data transfer.

The burst transfers to the HPI cannot occur at the full 132M bytes/s PCI data rate because the 'C6201 HPI is not a 32-bit-wide, synchronous interface. The HPI presents only a 16-bit interface; hardware handshaking via its ready signal requires multiple PCI clocks per 32-bit data transfer.

Because the 'C6201 is not 5-V signal tolerant, the S5933 outputs are translated to 3.3-V-compatible signals using two TI 'CBTD3384 buffer devices.

1.6.5 PCI Master Support

The 'C62x McEVM supports bus mastering of the PCI bus, enabling the McEVM to take control of the PCI bus and transfer data between the host and McEVM memory. The S5933 PCI controller handles the bus mastering transfers on the PCI bus. Transfers can be driven directly by 'C6201 software or automatically in the background with the DMA controller. This bus mastering support allows data transfers to be under McEVM control without continuous host intervention. The host does need to get involved initially to configure the S5933's PCI address and transfer counters and initiate the transfer, but it is not involved at all in the actual data transfers. The host initialization is required because only the host knows the address of its, and other PCI devices', memory buffers.

The S5933 supports bus master transfers by using two on-chip FIFOs for read and write transfers between the ’C6201 and the PCI bus. Each FIFO is 32 bits wide and 8 words deep. These FIFOs are addressed asynchronously from the DSP-side because the PCI controller and the DSP operate at different rates. The S5933 provides two FIFO flags that indicate the status of each FIFO (RDEMPTY/WRFULL). These FIFO flags are used to control the flow of data to and from the DSP.

The S5933 registers and FIFOs are memory mapped into the DSP’s CE1 asynchronous memory space, along with the MVIP FMIC and T1/E1 transceiver control registers, DSP memory-mapped CPLD registers, and asynchronous expansion memory. State machines in the CPLD manage the interface between the ’C6201 and the S5933 read and write FIFOs. They monitor S5933 FIFO flags and accesses by the DSP to the S5933 FIFO at address 0x1310000 (0x1710000), and generate the external interrupts to the DSP to control the data transfers. The FIFOs must be accessed using the dedicated memory-map address, rather than the PCI add-on register offset, for the external interrupts to be generated.

Master read transfer support and master write transfer support are independently enabled by the DSP by setting two bits in the CPLD’s memory-mapped FIFOSTAT register. When the PCIMREN bit (bit 1) is set, the CPLD generates EXT_INT5 interrupts to the DSP whenever data is available to be read in the PCI controller’s read FIFO. When the PCIMWEN bit (bit 0) is set, the CPLD generates EXT_INT6 interrupts to the DSP whenever the write FIFO is not full and can accept data. Typically, the DSP triggers DMA read and write transfers on these interrupts for the most efficient transfers. However, the FIFO flags in the CPLD’s FIFOSTAT register or the interrupts themselves can also be polled to drive the data transfers. The PCIMREN and PCIMWEN bits must be disabled at the completion of a bus master transfer and reenabled before the next one begins.

Both the host and DSP software must perform certain initialization and control actions to configure the ’C62x McEVM for PCI bus master transfers. The host and DSP software actions can be performed independently and do not require any synchronization. Depending on the application, it may be necessary for the host software to notify the DSP software about the DSP memory space source/destination address and number of words to be transferred. The PCI bus mastering operation begins only when both sides have completed their actions.

The following subsections identify the actions that must be performed by the host and DSP software to perform PCI bus master transfers.

1.6.5.1 Host Software Actions

The ’C62x McEVM is factory-configured for PCI (host) initiated bus master transfers. This means that only the host software can enable, control, and monitor the S5933 PCI controller’s FIFO bus master on the PCI bus. The McEVM uses the PCI-initiated configuration because only the host software knows the physical memory addresses of its memory, as well as other PCI devices’ memory. For PCI-initiated bus mastering, the host software must complete certain actions to set up the FIFO bus mastering.

❑ **Define interrupt capabilities.** The host can be interrupted independently at the end of read and write transfers when the transfer counters reach 0. The S5933 INTCSR register’s bits 14 and 15 must be set accordingly.

❑ **Reset FIFO flags.** The FIFO state must be initialized to empty by resetting the FIFO flags. The S5933 MCSR register’s bits 25 and 26 can be set to reset these flags.

❑ **Define FIFO management scheme.** The FIFOs must be configured for the desired times that the S5933 should request mastering the PCI bus. For reads, the PCI bus can be requested when there are at least one or at least four vacant FIFO locations. For writes, the PCI can be requested when there are at least one or at least four filled FIFO locations. The S5933 does not have to request the bus as often when four or more data words are transferred during each bus grant period; using this FIFO management scheme improves throughput in most applications. The FIFO management scheme is selected by initializing the S5933’s MCSR register’s bits 9 and 13.

❑ **Define PCI-to-McEVM and McEVM-to-PCI priority.** It is recommended that write versus read priority be set equal by setting bits 8 and 12 of the S5933’s MCSR register.

❑ **Define transfer source/destination address.** The start of the PCI-side source address must be written to the S5933’s MRAR register for bus master reads, and the destination address must be written to the MWAR register for bus master writes. This address is the system’s physical memory address that is presented on the PCI bus. This address must be on a 32-bit word address (lower two address bits 0).

❑ **Define transfer byte count.** The S5933’s MWTC and MRTC registers must be initialized with the number of bytes to be transferred for bus master writes and bus master reads, respectively. The number of bytes must be a multiple of four because the ’C62x McEVM only supports 32-bit, bus master data transfers.

- ❑ **Enable bus mastering.** After the other actions have been done, the read and write bus master operations can be independently enabled by setting bits 10 and 14 of the S5933's MCSR register. This enables the S5933's hardware to begin requesting the PCI bus to move data to and from its FIFOs.

- ❑ **Provide DSP software with data transfer information.** In most applications, the host software must notify the DSP software of the data transfer information, such as the DSP memory space source or destination address and the number of 32-bit words to be transferred. This allows the DSP software to initialize its DMA controller to handle the transfers from the S5933 to its memory space properly. This data transfer information can be sent to the DSP via the HPI or, more commonly, through the mailbox.

- ❑ **Provide application-specific response to end-of-transfer interrupt.** If interrupts were enabled, the host software must provide an interrupt service routine to handle PCI interrupts. The host software can perform an application-specific action in response to the end-of-transfer interrupt generated by the S5933. One example action is to prepare for the next data transfer.

The DSP must also perform certain actions in order for the bus mastering to begin.

1.6.5.2 DSP Software Actions

The ’C62x McEVM’s DSP software is responsible for moving the data between its memory space and the S5933 PCI controller’s bidirectional FIFO to complete the data transfers. The DSP typically uses DMA to handle the reading and writing of the data from and to the S5933’s FIFO. Optionally, the DSP software can perform this data movement itself using an interrupt service routine or by polling the status of the interrupts or FIFO flags themselves.

The ’C62x McEVM uses EXT_INT5 to control read bus master transfers from the S5933 read FIFO to the DSP memory space and EXT_INT6 to control write bus master transfers from the DSP memory space to the S5933 write FIFO. Logic in the CPLD, when explicitly enabled by the DSP software, monitors the FIFO flags and DSP accesses to the S5933 FIFO and controls the interrupts to the DSP.

The DSP software must perform the following actions to support PCI bus mastering:

❑ **Initialize ’C62x DMA control registers.** The primary DMA control register must be configured for incrementing address modification, 32-bit elements, and split-mode disabled. The transfers must be synchronized for reads or writes by initializing the RSYNC and WSYNC bits appropriately. For bus master reads, read synchronization based on EXT_INT5 must be selected. For bus master writes, write synchronization based on EXT_INT6 must be selected.

The secondary DMA control register can be configured to interrupt the DSP upon the transfer completion.

❑ **Initialize ’C62x DMA source/destination address register.** The source address register must be initialized for bus master writes, and the destination address register must be initialized for bus master reads.

❑ **Initialize ’C62x DMA transfer counter register.** The transfer counter register’s FRAME COUNT and ELEMENT COUNT values must be set to select the number of frames and 32-bit elements that are to be transferred. This value must match the number of transfers that the host software indicates.

❑ **Start the ’C62x DMA controller.** The DMA controller can be started by writing 01b to the START bits of the primary DMA control register.

❑ **Enable CPLD master transfer support.** Before bus master transfer interrupts are generated to the DSP, the respective CPLD control bits must be set to 1 as needed. For bus master writes, the CPLD’s FIFOSTAT PCIMWEN bit must be set to 1. For bus master reads, the PCIMREN bit must be set to 1. These enables activate a state machine in the CPLD that generates the external interrupts to the DSP.

❑ **Provide application-specific response to end-of-transfer interrupt.** If interrupts were disabled, the DSP software must provide an interrupt service routine to handle internal DMA block transfer complete interrupts. The DSP software can perform an application-specific action in response to the interrupt generated internal to the DSP. One example action is to prepare for the next data transfer.

❑ **Disable CPLD master transfer support.** To disable the bus master interrupts to the DSP and put the CPLD state machine in an idle state, the PCI bus master enable bits in the CPLD’s FIFOSTAT register must be cleared to 0. The bus master enable bits must be disabled before another master transfer is started.

1.7 JTAG Emulation

The McEVM provides embedded JTAG emulation, which is accessible via the PCI bus, as well as support for an external XDS510 emulator. The selected JTAG method is user configurable via the DIP switches when the board is operated outside the PC or via the software switches when it is in the PC.

The TI SN74ACT8990 JTAG test bus controller (TBC) provides memory-mapped control of the ’C6201’s JTAG interface. This allows the ’C62x McEVM C source debugger to be used with the McEVM without an external emulator.

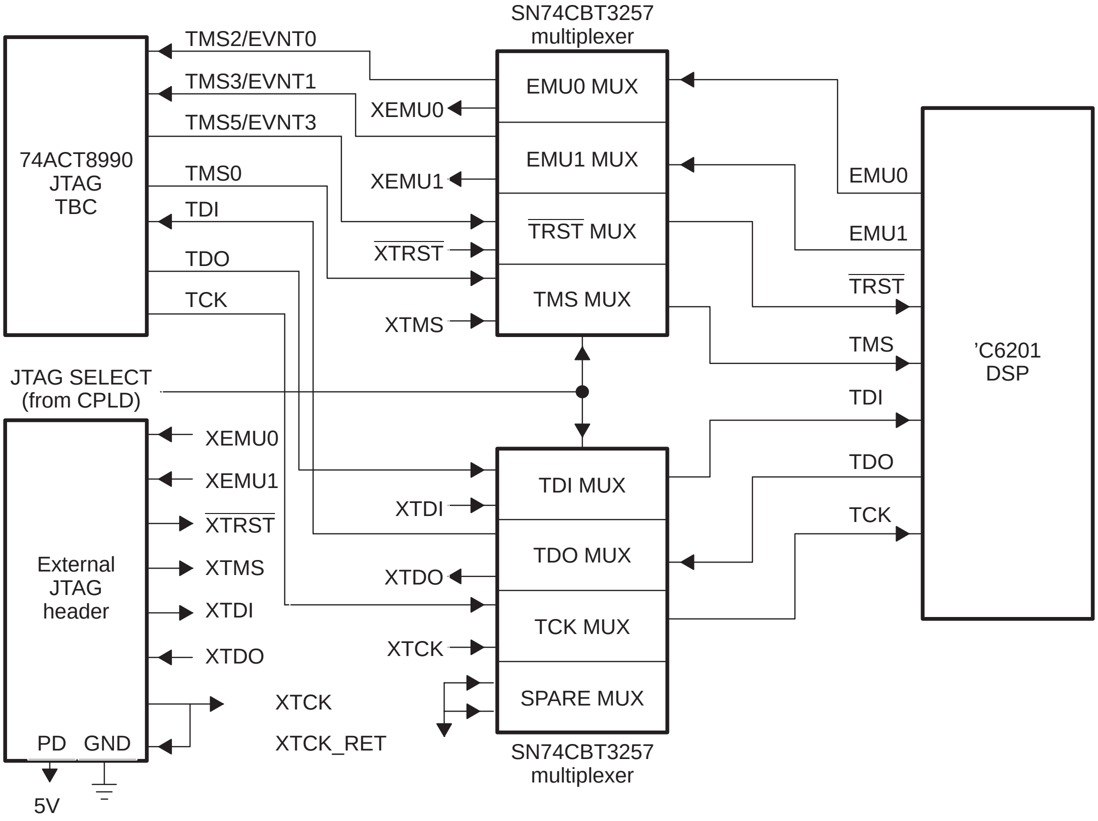
The ’C62x McEVM’s embedded emulation support provides several benefits:

- Emulation is supported without external cabling, monitor software, or consumption of user resources.
- Easy access to the ’C6201 supports high-level language (HLL) debuggers, factory testing, and field diagnostics.
- System boot ROMs are not needed. The host can download all necessary program and data information through the emulation port.

The TBC is presented to the PC host software as 24 memory-mapped registers. Each register is mapped at 32-bit (DWORD) address boundaries, but only the lower 16 bits of data words are connected to the 16-bit TBC device. The PCI controller’s pass-through interface is used to access the slave TBC. The CPLD includes a state machine that manages the S5933-to-TBC data transfers via the pass-through interface.

A 14-pin (two rows of seven pins) header on the McEVM (J10) supports an external XDS510 or XDS510WS emulator connection. This connection is required for debugging the McEVM outside of a PC. Two TI CBT quad 2:1 multiplexer devices (SN74CBT3257) are used to provide the 5-V to 3.3-V translation required from the TBC to the ’C6201 and the selection between internal and external JTAG emulation. The JTAG interface to the ’C6201 consists of seven signals: TMS, TDO, TDI, TCLK, $\overline{\text{TRST}}$, EMU1, and EMU0. The two quad multiplexer devices are required to switch and translate seven signals. The use of the CBT devices also allows 3.3- or 5-V external emulator connections. Figure 1–7 shows how the DSP’s JTAG signals are selected between the TBC and the external JTAG header using the CBT multiplexers.

Figure 1–7. JTAG Emulation Selection



The TBC does not directly provide pins dedicated for the EMU0, EMU1, and TRST signals that are present on the external JTAG header. However, the TBC can control and monitor these signals using the available TMSx pins. The TMS2 and TMS3 signals monitor the EMU0 and EMU1 signals from the ’C6201, respectively. The TMS5 signal controls the DSP’s TRST signal. The host emulation software monitors and controls these signals via the TBC’s PCI interface.

You can select between internal and external JTAG emulation via DIP switch SW2–9 during external operation or via a software switch for internal PCI operation. The selection controls the multiplexers’ select input signal accordingly.

A 10.368-MHz JTAG clock (TCK) is provided by the XDS510 or XDS510WS when external JTAG emulation is selected. The McEVM provides a 16.625-MHz clock when the JTAG TBC is used for embedded emulation. The 16.625-MHz clock is derived from the McEVM’s 33.25-MHz oscillator output using a single SN74F74 D-type flip-flop device configured for a divide-by-2 operation.

1.8 Programmable Logic

The ’C62x McEVM uses a CPLD (Altera™ part number EPM7256S) to implement the board’s required glue logic and to provide control and status interfaces for both host and DSP software. The CPLD provides the following functions:

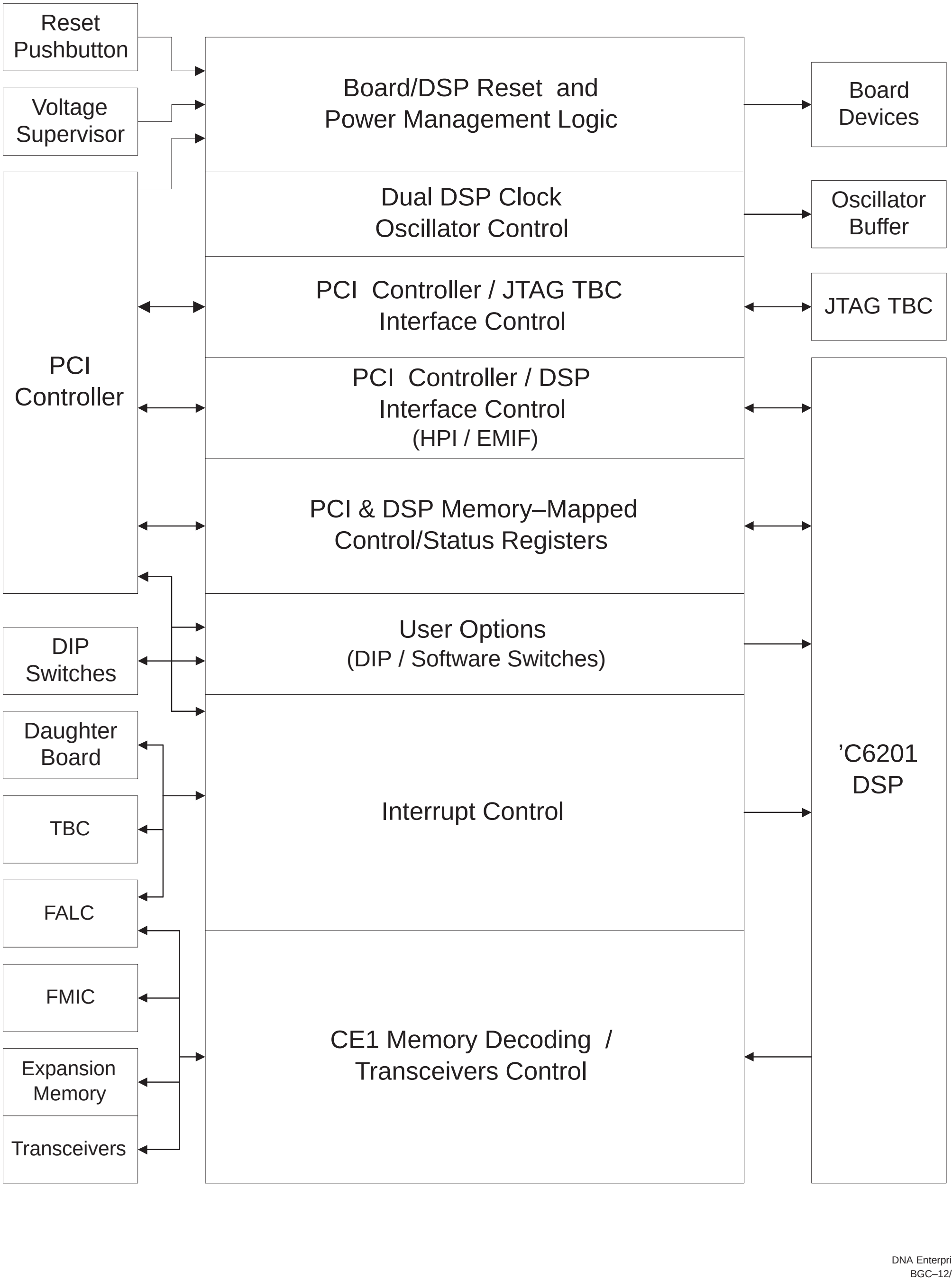
- ❑ Reset control
- ❑ Power management
- ❑ Dual DSP clock oscillator control
- ❑ PCI controller/DSP interface control
- ❑ PCI memory-mapped board control/status registers
- ❑ DSP memory-mapped control/status registers
- ❑ PCI and DSP interrupt control
- ❑ CE1 memory decoding
- ❑ Data transceivers control
- ❑ User options control
- ❑ Dual hardware semaphores

The EPM7256S CPLD is a 208-pin plastic quad flat pack (PQFP) device that provides 5000 usable gates, 160 user I/O pins, and a 10-ns pin-to-pin delay. The device is EEPROM-based and is in-system programmable via a dedicated JTAG interface presented as a 10-pin header on the McEVM. This header is a factory option that is not installed.

The EPM7256S uses 5 V for internal operation and input buffers and 3.3 V for output drivers. This provides an optimal design with fast internal speed and the ability to interface with both 3.3- and 5-V devices on the McEVM.

Figure 1–8 provides an overview of the CPLD’s functions and their associated interfaces.

Figure 1–8. ’C2x EVM CPLD Interfaces and Functions


Reset Pushbutton
Voltage Supervisor
PCI Controller
DIP Switches
Daughter Board
TBC
FALC
FMIC
Expansion Memory
Transceivers
Board/DSP Reset and Power Management Logic
Dual DSP Clock Oscillator Control
PCI Controller / JTAG TBC Interface Control
PCI Controller / DSP Interface Control (HPI / EMIF)
PCI & DSP Memory–Mapped Control/Status Registers
User Options (DIP / Software Switches)
Interrupt Control
CE1 Memory Decoding / Transceivers Control
Board Devices
Oscillator Buffer
JTAG TBC
’C6201 DSP
DNA Enterprises, Inc.
BGC–12/21/97

1.8.1 Reset Control

The CPLD works in tandem with a Maxim MAX708S voltage supervisor to provide several types of reset signals on the ’C62x McEVM.

The McEVM supports several methods to reset the board, DSP, JTAG TBC, and daughterboard. There are five reset sources:

- ❑ Power-up and undervoltage resets from the voltage supervisor
- ❑ Manual pushbutton reset
- ❑ PCI system reset from the PCI controller
- ❑ Daughterboard reset under DSP software control
- ❑ DSP and TBC resets under host software control

The MAX708S voltage supervisor asserts an active low reset to the CPLD whenever the 3.3-V supply is below 3.0 V, such as during power up and brownout conditions. Additionally, the voltage supervisor supports an external reset control signal generated by the CPLD that forces a board reset whenever the reset pushbutton is pressed or a PCI system or software reset is received. The MAX708S provides an output signal that indicates when the 1.8-V or 2.5-V supply has an undervoltage condition that the CPLD uses to control the DSP reset. The DSP is held in reset whenever either its core or I/O voltage is below a defined threshold.

The S5933 PCI controller provides a SYSRST# output that causes a reset whenever the PCI bus is reset, such as when a system is rebooted with Ctrl-Alt-Del or when the host software sets a bit in its MCSR register. The SYSRST# signal provides an automatic reset signal as well as a software-controlled reset control, similar to the voltage supervisor’s support for automatic and manual reset.

The DSP software has access to a memory-mapped CPLD register bit that directly controls the reset to the daughterboard. This daughterboard reset (XRESET#) is asserted low during reset but defaults to an inactive high upon release of reset. This provides a reset to the daughterboard whenever the McEVM is in reset, but it does not hold the daughterboard in reset. Subsequent control over the daughterboard reset is exclusively under DSP software control.

Both the DSP and JTAG TBC can be individually reset under host software control. PCI memory-mapped CPLD register bits allow the host software to directly control the reset signals to both the ’C6201 DSP and JTAG TBC. This is useful when doing controlled booting such as an HPI boot, performing a simple DSP reset operation, or putting the TBC in a known state. During board reset, the DSP and JTAG TBC are also held in reset. The DSP is also forced into reset whenever the board is in reset or its core voltage is not within specification. When the board reset is released, the two register bits default to not active so that the DSP and TBC are not held in reset in the external environment. The host software-controlled DSP and TBC resets are only available with internal PCI operation.

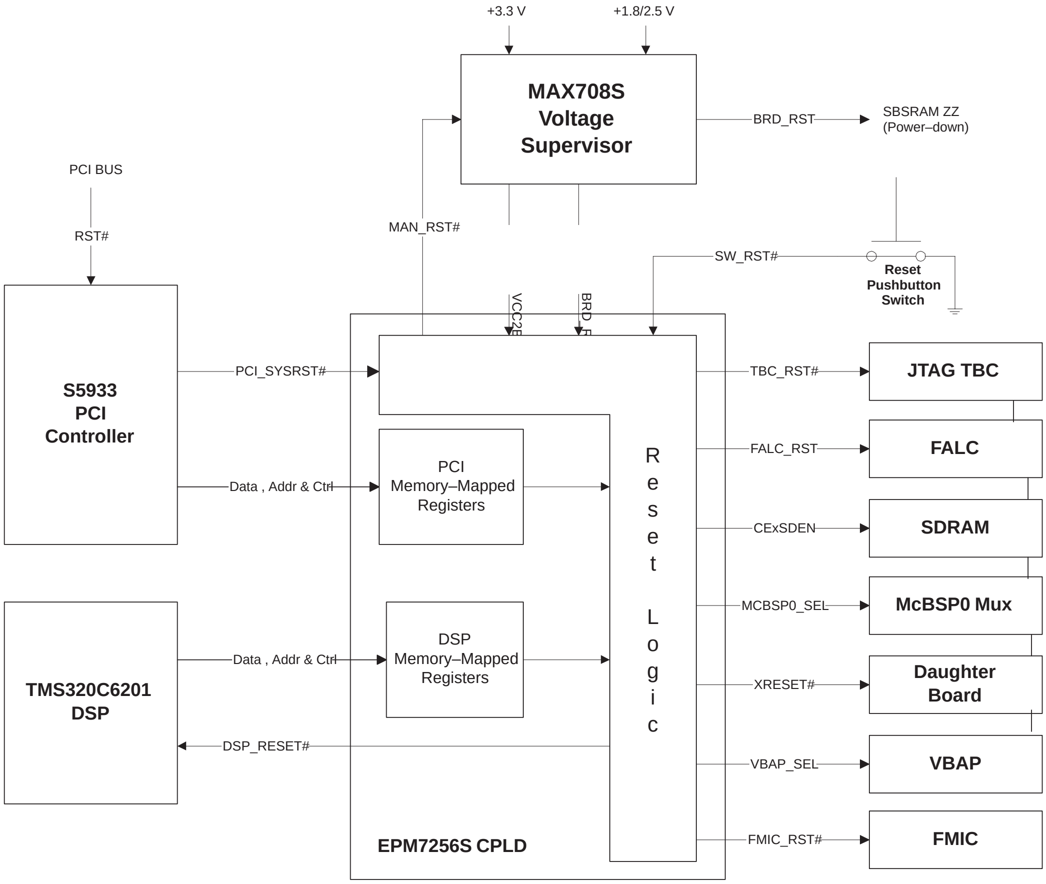
The board reset is produced when either the MAX708S indicates a reset (power up, undervoltage condition, or manual switch reset) or the PCI controller indicates a reset (PCI reset or manual software reset). The MAX708S provides two board reset signals: one active low and the other active high. The active-low board reset is routed to the CPLD, which directly controls the reset to the DSP, JTAG TBC, and daughterboard. The active-high board reset from the MAX708 is routed directly to the SBSRAM snooze (ZZ) input to put the device in an inactive, low-power state. When the McEVM board is held in reset, it cannot respond to host PCI accesses.

The PCI bus can only reset the PCI controller at power up and whenever the system is reset. The BIOS only assigns the memory region addresses and the host interrupt upon reset of the host PC.

In the PCI environment, the McEVM software drivers can put several board devices in a low-power state by holding the board in reset. The PCI specification recommends that PCI boards that dissipate more than 10 watts implement a default power reduction mode. This can be supported under driver control.

Figure 1–9 summarizes the ’C62x McEVM’s reset configuration.

Figure 1–9. Reset Configuration



1.8.2 Power Management

The *PCI Local Bus Specification Revision 2.1* recommends that PCI boards that can dissipate more than 10 watts under full operation have a power-saving state. The ’C62x McEVM CPLD includes logic that provides a power management capability that does not require additional devices. Various devices on the McEVM can be placed in a low-power state to reduce power consumption.

The McEVM’s power management feature is activated when the board is held in reset by the host software. This is achieved when the host software asserts the PCI controller’s SYSRST# signal (setting bit 24 of the S5933 PCI controller’s MCSR register).

When the McEVM board is held in reset, CPLD logic directly controls devices to power them down or put them into a low-power state. The 'C6201 DSP is held in reset, which forces it to power down. The SBSRAM is powered down by asserting its ZZ sleep input signal. The SBSRAM is disabled and put into a low-power mode by deasserting its CKE clock enable input signal. Both an external reset and the DSP's power-down indication are provided to the expansion peripheral interface to enable power-down support on a daughterboard. Other devices on the board, such as the buffers, are indirectly placed into a low-power mode, since their inputs are static when the other devices are inactive. Table 1–11 lists the McEVM devices that are directly controlled by CPLD logic and the control signals that are used to implement the power management feature of each device.

Table 1–11. Power Management Device Control Summary

Device	Power Management Control Signals
DSP	$\overline{\text{RESET}}$ = 0
SBSRAM	ZZ = 1
SDRAM	CKE = 0

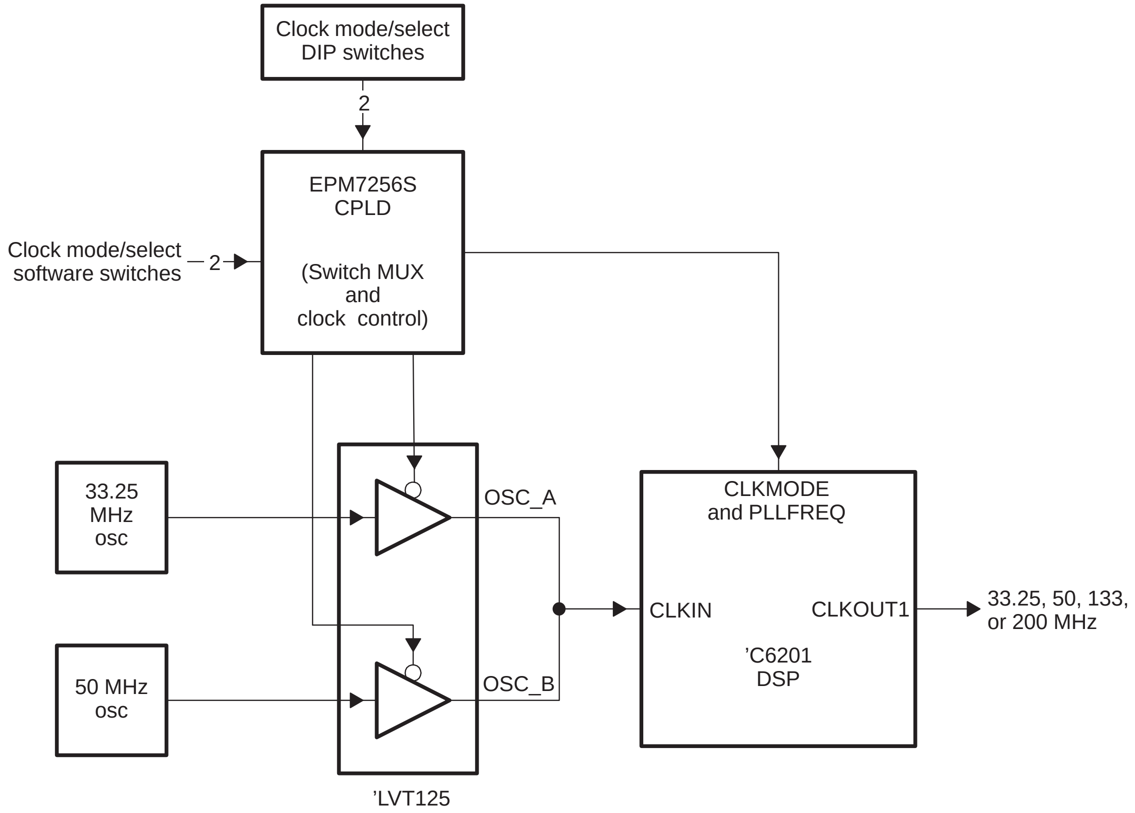
Daughterboard	XRESET = 0,
	DSP_PD = 1

1.8.3 Dual DSP Clock Oscillator Control

The 'C62x McEVM includes dual DSP clock oscillator support to enable operation at the 200-MHz core clock rate with one-half-rate SBSRAM timing or at the full 133-MHz SBSRAM clock rate. You have the flexibility to select the clock rate that provides the best performance for your application. This clock selection can be made with the McEVM DIP switches or from host software using the support software switches. For external operation, only the DIP switches are used to select the clock. For internal (PCI) operation, the DIP switches are selected for control by default unless overridden by host software using the software switches.

The user clock selection is used by CPLD logic to control two clock buffer enables. Each clock oscillator's output is connected to an 'LVT125 buffer that has independent output enables. The two clock buffer outputs are connected together to drive the DSP's CLKIN input. To provide the 2:1 clock selection function and ensure that there is no contention between the oscillators, the CPLD logic performs a "break-before-make" function that ensures that only one clock oscillator is driving the DSP's CLKIN input. Figure 1–10 summarizes the DSP clock selection configuration.

Figure 1–10. DSP Clock Selection Configuration



1.8.4 PCI Controller/JTAG TBC Interface Control

The CPLD includes a PCI add-on bus state machine that monitors and controls the interfaces between the S5933 PCI controller and the PCI memory-mapped devices on the McEVM, including the JTAG TBC.

Whenever the S5933 indicates either a PCI read or write TBC access, the state machine acknowledges it by asserting S5933 and TBC control signals required to complete the data transfer. The lower 16 bits of the S5933 add-on data bus are connected to the TBC data bus interface. The CPLD latches the TBC register address during the PCI address phase, and the state machine asserts the TBC read and write strobes to enable data transfers.

1.8.5 PCI Controller/DSP Interface Control

The CPLD’s add-on bus state machine supports PCI transfers with the DSP’s HPI, as well as DSP EMIF transfers with the S5933 PCI controller. These capabilities allow the host and DSP software to communicate with one another. The host software has read/write access to the DSP’s memory space and can exchange messages with the DSP software using the S5933’s mailboxes. The DSP software has read/write access to the host PC’s memory space and can also exchange messages with the host software using S5933’s mailboxes. The S5933’s bidirectional FIFOs can also be used to pass data between the host and DSP software, although they are typically only used for bus master transfers.

The DSP HPI can be accessed by the host software with BAR3 and BAR4 PCI accesses. Whenever the S5933 indicates either a PCI read or write HPI access, the state machine acknowledges it by asserting S5933 and HPI control signals required to complete the data transfer. The lower 16 bits of the S5933 add-on data bus are connected to the HPI data bus interface, with TI ’CBT devices used to translate from 5- to 3.3-V signals. The internal byte-lane switching feature of the S5933 allows the low and high words of the 32-bit PCI data transfers to be output on the lower 16 bits of the add-on bus during the low and high word data transfer times to the HPI. The CPLD latches the HPI register address during the PCI address phase, and the state machine asserts the HPI control signals to enable data transfers.

CPLD output pins are shared to support both the TBC address and HPI control signals. Because an access can only be directed to one of the two at any one time, the outputs can be used for both interfaces and controlled as required for the different accesses. This is possible since the TBC address signals are don’t cares unless the TBC read/write strobes are asserted, and the HPI control signals are don’t cares unless the HPI chip and data select signals are asserted.

HPI data transfers are 16 bits wide, so a 32-bit data transfer between the host and DSP requires two HPI data transfers. The CPLD state machine manages each 16-bit data transfer by controlling the appropriate byte enable and other control signals. The state machine includes monitoring of the HPI ready output signal to determine when the next write can proceed and when read data is available.

Because the state machine is synchronous to the 33-MHz PCI clock, there is a minimum of 30 ns between control signal transitions. Therefore, the maximum transfer rate between the host and DSP is defined by the required HPI transfer protocol to transfer two 16-bit words and the 30-ns state machine clock period.

CPLD arbitration logic allows both PCI and DSP EMIF transfers to share the common PCI add-on bus. The add-on state machine controls DSP EMIF accesses to the S5933 controller's registers and FIFOs. Because the state machine may be in the process of transferring data between the S5933 and another device when the DSP attempts to access the S5933, the state machine provides arbitration logic that holds off the EMIF access via the DSP's ARDY input until the current add-on transfer is finished. Burst transfers are finished as soon as possible to allow EMIF accesses to have priority over the add-on bus. Logic is also included that disables the EMIF data bus connection to the add-on bus while the add-on bus is being used for other transfers to prevent data bus contention. When an EMIF access to the S5933 is taking place, PCI transfers are held until the add-on bus is released.

The CPLD add-on bus state machine controls the add-on bus control signals during EMIF accesses to the S5933 registers and FIFOs. When the EMIF is granted the add-on bus, the state machine latches the register address from the DSP's address lines and presents them on the add-on address bus. The other add-on bus control signals are also asserted by the state machine to provide data to the DSP EMIF data bus or to latch the data into the selected S5933 register or FIFO.

There is a 32-bit data interface between the DSP EMIF and the S5933 PCI controller. The DSP's EMIF 32-bit data bus is buffered with 'ALVCH16245 and 'CBTD3384 devices to isolate it from the PCI add-on bus and to provide voltage translation between 5 V and 3.3 V.

1.8.6 PCI Memory-Mapped Board Control/Status Registers

The CPLD's add-on bus state machine supports PCI transfers with eleven memory-mapped board control and status registers that are also implemented in the CPLD. These registers provide the host with the following capabilities:

- ❑ Interrupt the DSP (nonmaskable interrupt)
- ❑ Enable interrupts from the DSP and TBC
- ❑ Reset the DSP and TBC
- ❑ Monitor status of McEVM devices
- ❑ Read the user-option DIP switches
- ❑ Select the user options via software switches
- ❑ Read the selected DSP options
- ❑ Read the CPLD revision number
- ❑ Utilize two hardware semaphores

Table 1–12 summarizes the PCI memory-mapped CPLD registers.

Table 1–12. PCI Memory-Mapped CPLD Registers

Address	Name	Description	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BAR2 + 0x00	CNTL	Control	DSPNMI RW 0 (no NMI)	TBCINTEN RW 0 (disabled)	HINTEN RW 0 (disabled)	TBCRDY R –	TBCINT R –	DSPHINT R –	TBCRST RW 0 (no reset)	DSPRST RW 0 (no reset)
BAR2 + 0x04	STAT	Status	– – –	VCC2BAD R –	– – –	DSPPD R –	XCNTL1 R –	XCNTL0 R –	USERLED1 R –	USERLED0 R –
BAR2 + 0x08	SWOPT	SW switch – options	– – –	H_CLKMODE RW 0 (X4)	H_CLKSEL RW 0 (osc A)	H_ENDIAN RW 0 (little)	H_JTAGSEL RW 0 (ext)	H_USER2 RW 0	H_USER1 RW 0	H_USER0 RW 0
BAR2 + 0x0C	SWBOOT	SW switch – boot mode	SWSEL RW 0 (DIP switch)	– – –	– – –	H_BMODE4 RW 0	H_BMODE3 RW 0	H_BMODE2 RW 1	H_BMODE1 RW 0	H_BMODE0 RW 1
BAR2 + 0x10	DIPOPT	DIP switch – options	– – –	S_CLKMODE R –	S_CLKSEL R –	S_ENDIAN R –	S_JTAGSEL R –	S_USER2 R –	S_USER1 R –	S_USER0 R –
BAR2 + 0x14	DIPBOOT	DIP switch – boot mode	– – –	– – –	– – –	S_BMODE4 R	S_BMODE3 R	S_BMODE2 R	S_BMODE1 R	S_BMODE0 R
BAR2 + 0x18	DSPOPT	DSP – options	– – –	CLKMODE R –	CLKSEL R –	LENDIAN R –	JTAGSEL R –	USER2 R –	USER1 R –	USER0 R –
BAR2 + 0x1C	DSPBOOT	DSP – boot mode	– – –	– – –	– – –	BMODE4 R –	BMODE3 R –	BMODE2 R –	BMODE1 R –	BMODE0 R –
BAR2 + 0x20	CPLDREV	CPLD revision	CREV7 R –	CREV6 R –	CREV5 R –	CREV4 R –	CREV3 R –	CREV2 R –	CREV1 R –	CREV0 R –
BAR2 + 0x24	SEM0	– – –	– – –	– – –	– – –	– – –	– – –	– – –	– – –	PCISEM0 RW –
BAR2 + 0x28	SEM1	– – –	– – –	– – –	– – –	– – –	– – –	– – –	– – –	PCISEM1 RW 0

The following subsections describe each of the PCI memory-mapped CPLD registers.

Note:

All register bits are active high (1) for consistency and ease of use. For example, to reset the DSP, bit 0 of the CNTL register must be set to a 1. Highlighted register values denote the power-up default values.

1.8.6.1 PCI CNTL Register (BAR2 + 0x00)

The CNTL register enables the host software to interrupt and reset the DSP, enable interrupts from the DSP, and monitor TBC and DSP status. The host can be interrupted whenever the TBC or DSP host interrupts are asserted, or it can poll the status of these interrupt signals directly. The nonmaskable interrupt (NMI) and reset bits must be manually toggled to assert and deassert the respective signals. Table 1–13 summarizes the function of each bit in the CNTL register. Highlighted register values denote the power-up default values.

Table 1–13. PCI CNTL Register Bit Definitions

Bit	Name	Access	Description
7	DSPNMI	RW	Controls DSP’s NMI (**0 = no NMI**, 1 = assert NMI)
6	TBCINTEN	RW	TBC interrupt enable (**0 = disable TBC interrupt**, 1 = enable TBC interrupt)
5	HINTEN	RW	DSP host interrupt enable (**0 = disable host interrupt**, 1 = enable host interrupt)
4	TBCRDY	R	TBC ready status (0 = TBC not ready, 1 = TBC ready)
3	TBCINT	R	TBC interrupt status (0 = no TBC interrupt, 1 = TBC interrupt)
2	DSPHINT	R	DSP host interrupt status (0 = no host interrupt, 1 = host interrupt)
1	TBCRST	RW	TBC hardware reset (**0 = no TBC reset**, 1 = TBC reset)
0	DSPRST	RW	DSP hardware reset (**0 = no DSP reset**, 1 = DSP reset)

1.8.6.2 PCI STAT Register (BAR2 + 0x04)

The STAT register enables the host software to monitor the DSP core voltage, DSP power down, daughterboard control, and user-defined LED status. Table 1–14 summarizes the function of each bit in the STAT register.

Table 1–14. PCI STAT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	VCC2BAD	R	DSP core voltage status (0 = core voltage OK, 1 = core voltage bad)
4	DSPPD	R	DSP power-down status (0 = DSP not in power down, 1 = DSP in power down)
3	XCNTL1	R	Control signal from DSP to daughterboard (0 = TTL low, 1 = TTL high)
2	XCNTL0	R	Control signal from DSP to daughterboard (0 = TTL low, 1 = TTL high)
1	USERLED1	R	User LED 1 status (0 = LED extinguished, 1 = LED illuminated)
0	USERLED0	R	User LED 0 status (0 = LED extinguished, 1 = LED illuminated)

1.8.6.3 PCI SWOPT Register (BAR2 + 0x08)

The SWOPT register enables the host software to override the McEVM’s onboard user option DIP switches. This capability provides software switches, which allow the DSP options to be controlled without having to remove the PC’s cover. The values in this register are not used unless the SWSEL bit in the SWBOOT register is set to 1. Table 1–15 summarizes the function of each bit in the SWOPT register. Highlighted register values denote the power-up default values.

Table 1–15. PCI SWOPT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	H_CLKMODE	RW	Host clock mode (**0 = ×4 mode**, 1 = ×1 mode)
5	H_CLKSEL	RW	Host clock select (**0 = OSC_A = 33.25 MHz**, 1 = OSC_B = 50 MHz)
4	H_ENDIAN	RW	Host endian control (**0 = little endian**, 1 = big endian)
3	H_JTAGSEL	RW	Host JTAG selection (**0 = external XDS510**, 1 = onboard JTAG TBC)
2	H_USER2	RW	Host user-defined option 0 (**0 = on**, 1 = off)
1	H_USER1	RW	Host user-defined option 1 (**0 = on**, 1 = off)
0	H_USER0	RW	Host user-defined option 2 (**0 = on**, 1 = off)

1.8.6.4 PCI SWBOOT Register (BAR2 + 0x0C)

The SWBOOT register enables the host software to override the McEVM's boot mode DIP switches. This capability provides software switches, which allow the DSP boot mode to be controlled without having to remove the PC's cover. The values in this register do not get used unless the SWSEL bit is set to 1. Table 1–16 summarizes the function of each bit in the SWBOOT register. Highlighted register values denote the power-up default values.

Table 1–16. PCI SWBOOT Register Bit Definitions

Bit	Name	Access	Description
7	SWSEL	RW	Software switch select (**0 = DIP switches**, 1 = software switches)
6	–	–	–
5	–	–	–
4	H_BMODE4	RW	Host boot mode 4 (**0**)
3	H_BMODE3	RW	Host boot mode 3 (**0**)
2	H_BMODE2	RW	Host boot mode 2 (**1**)
1	H_BMODE1	RW	Host boot mode 1 (**0**)
0	H_BMODE0	RW	Host boot mode 0 (**1**)

1.8.6.5 PCI DIPOPT Register (BAR2 + 0x10)

The DIPOPT register enables the host software to read the McEVM's onboard user option DIP switches. Table 1–17 summarizes the function of each bit in the DIPOPT register.

Table 1–17. PCI DIPOPT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	S_CLKMODE	R	Switch clock mode (0 = ×4 mode, 1 = ×1 mode)
5	S_CLKSEL	R	Switch clock select (0 = OSC_A = 33.25 MHz, 1 = OSC_B = 50 MHz)
4	S_ENDIAN	R	Switch endian control (0 = little endian, 1 = big endian)
3	S_JTAGSEL	R	Switch JTAG selection (0 = external XDS510, 1 = onboard JTAG TBC)
2	S_USER2	R	Switch user-defined option 0 (0 = on, 1 = off)
1	S_USER1	R	Switch user-defined option 1 (0 = on, 1 = off)
0	S_USER0	R	Switch user-defined option 2 (0 = on, 1 = off)

1.8.6.6 PCI DIPBOOT Register (BAR2 + 0x14)

The DIPBOOT register enables the host software to read the McEVM's onboard boot mode DIP switches. Table 1–18 summarizes the function of each bit in the DIPBOOT register.

Table 1–18. PCI DIPBOOT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	S_BMODE4	R	Switch boot mode 4
3	S_BMODE3	R	Switch boot mode 3
2	S_BMODE2	R	Switch boot mode 2
1	S_BMODE4	R	Switch boot mode 1
0	S_BMODE4	R	Switch boot mode 0

1.8.6.7 PCI DSPOPT Register (BAR2 + 0x18)

The DSPOPT register enables the host software to read the DSP options, which may either be from the DIP switches or the software switches, depending on which is selected. Table 1–19 summarizes the function of each bit in the DSPOPT register.

Table 1–19. PCI DSPOPT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	CLKMODE	R	Clock mode (0 = ×1 mode, 1 = ×4 mode)
5	CLKSEL	R	Clock select (0 = OSC_A = 33.25 MHz, 1 = OSC_B = 50 MHz)
4	LENDIAN	R	Endian control (0 = big endian, 1 = little endian)
3	JTAGSEL	R	JTAG selection (0 = external XDS510, 1 = onboard JTAG TBC)
2	USER2	R	User-defined option 0 (0 = on, 1 = off)
1	USER1	R	User-defined option 1 (0 = on, 1 = off)
0	USER0	R	User-defined option 2 (0 = on, 1 = off)

1.8.6.8 PCI DSPBOOT Register (BAR2 + 0x1C)

The DSPBOOT register enables the host software to read the DSP's boot, which can either be from the DIP switches or the software switches, depending on which is selected. Table 1–20 summarizes the function of each bit in the DSPBOOT register.

Table 1–20. PCI DSPBOOT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	BMODE4	R	Boot mode 4
3	BMODE3	R	Boot mode 3
2	BMODE2	R	Boot mode 2
1	BMODE1	R	Boot mode 1
0	BMODE0	R	Boot mode 0

1.8.6.9 PCI CPLDREV Register (BAR2 + 0x20)

The CPLDREV register provides the revision of the McEVM's CPLD. This information may be useful in assisting technical support. The revision number can be up to eight bits in length. Table 1–21 summarizes the function of each bit in the CPLDREV register.

Table 1–21. PCI CPLDREV Register Bit Definitions

Bit	Name	Access	Description
7	CREV7	R	CPLD revision bit 7
6	CREV6	R	CPLD revision bit 6
5	CREV5	R	CPLD revision bit 5
4	CREV4	R	CPLD revision bit 4
3	CREV3	R	CPLD revision bit 3
2	CREV2	R	CPLD revision bit 2
1	CREV1	R	CPLD revision bit 1
0	CREV0	R	CPLD revision bit 0

1.8.6.10 PCI SEM0 Register (BAR2 + 0x24)

The SEM0 register provides a single semaphore flag that can be used to share devices or coordinate activities between the host and the DSP. Only bit 0 (PCI-SEM0) is valid in this register. A semaphore is requested by writing a ‘1’ to this bit. The bit is read back to determine if the semaphore was granted. If PCI-SEM0 is read back as ‘1’, then the host owns the semaphore. If it is read back as a ’0’, then the host does not own the semaphore. It is important to note that a semaphore request must be explicitly made each time, since pending requests are not supported. When the host wants to relinquish the semaphore, a ‘0’ should be written to the PCI SEM0 bit. Table 1–22 summarizes the function of each bit in the semaphore 0 register.

The DSP accesses the same semaphore as the host, but only the host or the DSP can own it at any one time. At reset, the semaphore is set to ’0’ on both sides, so neither one owns the semaphore.

Table 1–22. PCI SEM0 Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	–	–	–
3	–	–	–
2	–	–	–
1	–	–	–
0	PCISEM0	RW	Semaphore 0 Flag (For reads: 0=not owned, 1=owned; For writes: 0=relinquish, 1=request)

1.8.6.11 PCI SEM1 Register (BAR2 + 0x28)

The SEM1 register provides a single semaphore flag that can be used to share devices or coordinate activities between the host and the DSP. Only bit 0 (PCI-SEM1) is valid in this register. A semaphore is requested by writing a ‘1’ to this bit. The bit is read back to determine if the semaphore was granted. If PCI-SEM1 is read back as a ‘1’, then the host owns the semaphore. If it is read back as a ’0’, then the host does not own the semaphore. It is important to note that a semaphore request must be explicitly made each time, since pending requests are not supported. When the host wants to relinquish the semaphore, a ‘0’ should be written to the PCISEM1 bit. Table 1–23 summarizes the function of each bit in the semaphore 1 register.

Table 1–23. PCI SEM1 Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	–	–	–
3	–	–	–
2	–	–	–
1	–	–	–
0	PCISEM1	RW	Semaphore 1 Flag (For reads: 0=not owned, 1=owned; For writes: 0=relinquish, 1=request)

1.8.7 DSP Memory-Mapped Control/Status Registers

The ’C62x McEVM’s CPLD provides fourteen control and status registers that are memory-mapped into the DSP’s CE1 memory space. All registers are eight bits wide and are mapped into the least-significant byte of the EMIF data bus (ED[7:0]). The registers are mapped on DWORD address boundaries for little-endian mode, but, for big-endian mode, the registers are offset by 3 to access ED[7:0]. The memory-mapped DSP registers provide the DSP software to perform the following functions:

- Control the two user-defined LEDs
- Control and monitor the daughterboard
- Control DSP NMI
- Determine the interrupt status
- Determine the operating environment
- Read the user options DIP switches
- Read the selected DSP options
- Monitor and control the PCI controller FIFO
- Control the two SDRAM banks
- Use two hardware semaphores
- Control the MVIP FMIC and local stream connections
- Reset the FMIC and FALC devices
- Support FALC and daughterboard interrupts
- Control four T1/E1 status LED indicators
- Enable and select µ-Law and A-Law handset interfaces

Address decoding generates the clock and output enable controls for the registers. Register clock enables are generated whenever the specific register is being addressed. The rising edge of the EMIF $\overline{\text{AWE}}$ output is used to clock the lower eight bits of the EMIF data bus into the registers. Data is output on the lower eight bits of the EMIF data bus when the CPLD register space is read. This register space is allocated 64K bytes in the CE1 memory space, as shown in Table 1–2 and Table 1–3. The eight registers are addressed sequentially on DWORD address boundaries.

The lower eight bits of the EMIF data bus are buffered by an ’ALVCH16245 transceiver to provide the global data bus (GD[7:0]) that is connected to the CPLD.

Table 1–24 summarizes the fourteen DSP memory-mapped control and status registers implemented in the CPLD.

Table 1–24. DSP Memory-Mapped Control/Status CPLD Registers

Address	Name	Description	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
01780000 (01380000)	CNTL	Control	XCNTL1 RW 0 (inactive)	XCNTL0 RW 0 (inactive)	XRESET RW 0 (no reset)	NMIEN RW 0 (disabled)	– – –	SP0SEL RW 0 (DB)	LED1 RW 0 (off)	LED0 RW 0 (off)
01780004 (01380004)	STAT	Status	XSTAT1 R –	XSTAT0 R –	DBINT R –	DSPNMI R –	– – –	– – –	PCIINT R –	PCIDET R –
01780008 (01380008)	DIPOPT	DIP switch – options	– – –	S_CLKMODE R –	S_CLKSEL R –	S_ENDIAN R –	S_JTAGSEL R –	S_USER2 R –	S_USER1 R –	S_USER0 R –
0178000C (013C000C)	DIPBOOT	DIP switch – boot mode	– – –	– – –	– – –	S_BMODE4 R –	S_BMODE3 R –	S_BMODE2 R –	S_BMODE1 R –	S_BMODE0 R –
01780010 (01380010)	DSPOPT	DSP – options	– – –	CLKMODE R –	CLKSEL R –	LENDIAN R –	JTAGSEL R –	USER2 R –	USER1 R –	USER0 R –
01780014 (01380014)	DSPBOOT	DSP – boot mode	SWSEL R –	– – –	– – –	BMODE4 R –	BMODE3 R –	BMODE2 R –	BMODE1 R –	BMODE0 R –
01780018 (01380018	FIFOSTAT	PCI FIFO – status/control	– – –	– – –	PCIMRINT R –	PCIMWINT R –	RDEMPTY R –	WRFULL R –	PCIMREN RW 0 (disabled)	PCIMWEN RW 0 (disabled)
0178001C (0138001C)	SDCNTL	SDRAM control (CE2/CE3)	– – –	– – –	– – –	– – –	– – –	– – –	CE3_SDEN RW 1 (enable)	CE2_SDEN RW 1 (enable)
01780020 (01380020)	OSCB	Oscillator B Frequency (50 MHz)	FREQ7– R 0	FREQ7 R 0	FREQ7 R 1	FREQ7 R 1	FREQ7 R 0	FREQ7 R 0	FREQ7 R 1	FREQ7 R 0
01780024 (01380024)	SEM0	Semaphore 0	– – –	– – –	– – –	– – –	– – –	– – –	– – –	DSPSEM0 RW 0

Address	Name	Description	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
01780028 (01380028)	SEM1	Semaphore 1	– – –	– – –	– – –	– – –	– – –	– – –	– – –	DSPSEM1 RW 0
0178002C (0138002C)	FALC-CNTL	FALC/FMIC Control	MSEL3 RW 0	MSEL2 RW 0	MSEL1 RW 0	MSEL0– RW 0	LD31SWP RW 0	LD20SWP RW 0	FMICRST RW 0	FALCRST RW 0
01780030 (01380030)	INTCNTL	Interrupt Control	LTDBINT R –	LTFALCINT R –	DBINT R –	FALCINT R –	– – –	– – –	CDBINT RW 0	CFALCINT RW 0
01780034 (01380034)	MISC	Miscellaneous Control	LOOPLED RW 0	SYNCLED RW 0	REDLED RW 0	YELLED RW 0	MU_ALAW RW 0	VBAPEN RW 0	FMICERR R 0	– – –

The following subsections describe each of the fourteen DSP memory-mapped CPLD registers.

Note:

All register bits are active high (1) for consistency and ease of use. For example, to illuminate LED0, bit 0 of the CNTL register must be set to a 1. Highlighted register values denote the power-up default values.

1.8.7.1 DSP CNTL Register (0x01380000/0x01780000)

The CNTL register enables the DSP software to control the daughterboard, enable and select the NMI interrupt source, select the McBSP0 connection, and control the user-defined LEDs. Table 1–25 summarizes the function of each bit in the CNTL register. Highlighted register values denote the power-up default values.

Table 1–25. DSP CNTL Register Bit Definitions

Bit	Name	Access	Description
7	XCNTL1	RW	Control signal to daughterboard **(0 = TTL low,** 1 = TTL high)
6	XCNTL0	RW	Control signal to daughterboard **(0 = TTL low**, 1 = TTL high)
5	XRESET	RW	Daughterboard reset signal (**0 = no reset**, 1 = asserts active low reset)
4	NMIEN	RW	NMI interrupt enable (**0 = disable NMI to DSP**, 1 = enable NMI to DSP)
3	–	–	–
2	SP0SEL	RW	McBSP0 selection (**0 = daughterboard**, 1 = MVIP FMIC)
1	LED1	RW	User-defined LED #1 on top of board (**0 = extinguished**, 1 = illuminated)
0	LED0	RW	User-defined LED #0 on bracket (**0 = extinguished**, 1 = illuminated)

1.8.7.2 DSP STAT Register (0x01380004/0x01780004)

The STAT register enables the DSP software to monitor the daughterboard, interrupts, and PCI detection indicator. Table 1–26 summarizes the function of each bit in the STAT register.

Table 1–26. DSP STAT Register Bit Definitions

Bit	Name	Access	Description
7	XSTAT1	R	Status signal from daughterboard (0 = TTL low, 1 = TTL high)
6	XSTAT0	R	Status signal from daughterboard (0 = TTL low, 1 = TTL high)
5	DBINT	R	Daughterboard interrupt status (0 = no interrupt, 1 = interrupt)
4	DSPNMI	R	DSP NMI interrupt (0 = no NMI, 1 = NMI)
3	–	–	–
2	–	–	–
1	PCIINT	R	PCI interrupt status (0 = no PCI interrupt, 1 = PCI interrupt asserted)
0	PCIDET	R	PCI detection indicator (0 = external operation, 1 = PCI operation)

1.8.7.3 DSP DIPOPT Register (0x01380008/0x01780008)

The DIPOPT register enables the DSP software to read the DIP switch user options. Table 1–27 summarizes the function of each bit in the DIPOPT register.

Table 1–27. DSP DIPOPT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	S_CLKMODE	R	Switch clock mode (0 = ×4 mode, 1 = ×1 mode)
5	S_CLKSEL	R	Switch clock select (0 = OSC_A = 33.25 MHz, 1 = OSC_B = 50 MHz)
4	S_ENDIAN	R	Switch endian control (0 = little endian, 1 = big endian)
3	S_JTAGSEL	R	Switch JTAG selection (0 = external XDS510, 1 = onboard JTAG TBC)
2	S_USER2	R	User-defined switch 0 (0 = on, 1 = off)
1	S_USER1	R	User-defined switch 1 (0 = on, 1 = off)
0	S_USER0	R	User-defined switch 2 (0 = on, 1 = off)

1.8.7.4 DSP DIPBOOT Register (0x0138000C/0x0178000C)

The DIPBOOT register enables the DSP software to read the DIP switch boot mode selection. Table 1–28 summarizes the function of each bit in the DIPBOOT register.

Table 1–28. DSP DIPBOOT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	S_BMODE4	R	Switch boot mode 4 (see Table 1–44 on page 1-90 for valid values)
3	S_BMODE3	R	Switch boot mode 3
2	S_BMODE2	R	Switch boot mode 2
1	S_BMODE1	R	Switch boot mode 1
0	S_BMODE0	R	Switch boot mode 0

1.8.7.5 DSP DSPOPT Register (0x01380010/0x01780010)

The DSPOPT register enables the DSP software to read the actual DSP options. Table 1–29 summarizes the function of each bit in the DSPOPT register.

Table 1–29. DSP DSPOPT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	CLKMODE	R	Switch clock mode (0 = ×1 mode, 1 = ×4 mode)
5	CLKSEL	–	Switch clock select (0 = OSC_A = 33.25 MHz, 1 = OSC_B = 50 MHz)
4	LENDIAN	R	Switch endian control (0 = big endian, 1 = little endian)
3	JTAGSEL	R	Switch JTAG selection (0 = external XDS510, 1 = onboard JTAG TBC)†
2	USER2	R	User-defined switch 2 (0 = on, 1 = off)
1	USER1	R	User-defined switch 1 (0 = on, 1 = off)
0	USER0	R	User-defined switch 0 (0 = on, 1 = off)

† Bit 3 (JTAGSEL) is always 0 when the McEVM is not installed in a PCI slot.

1.8.7.6 DSP DSPBOOT Register (0x01380014/0x01780014)

The DSPBOOT register enables the DSP software to read the DSP boot mode selection. Table 1–30 summarizes the function of each bit in the DSPBOOT register.

Table 1–30. DSP DSPBOOT Register Bit Definitions

Bit	Name	Access	Description
7	SWSEL	R	Software switch select (0 = DIP switches, 1 = software switches)
6	–	–	–
5	–	–	–
4	BMODE4	R	Boot mode 4 (see Table 1–44 on page 1-90 for valid values)
3	BMODE3	R	Boot mode 3
2	BMODE2	R	Boot mode 2
1	BMODE1	R	Boot mode 1
0	BMODE0	R	Boot mode 0

1.8.7.7 DSP FIFOSTAT Register (0x01380018/0x01780018)

The FIFOSTAT register enables the DSP software to determine the FIFOs' empty/full status and enable PCI bus master external interrupts based on the FIFOs' status. Table 1–31 summarizes the function of each bit in the FIFOSTAT register. Highlighted register values denote the power-up default values.

Table 1–31. DSP FIFOSTAT Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	PCIMRINT	R	PCI master read interrupt (**0 = inactive**, 1 = active)
4	PCIMWINT	R	PCI master write interrupt (**0 = inactive**, 1 = active)
3	RDEMPTY	R	PCI controller read FIFO empty (0 = not empty, **1 = empty**)
2	WRFULL	R	PCI controller write FIFO full (**0 = not full**, 1 = full)
1	PCIMREN	RW	PCI master read enable (**0 = disable**, 1 = enable)
0	PCIMWEN	RW	PCI master write enable (**0 = disable**, 1 = enable)

1.8.7.8 DSP SDCNTL Register (0x0138001C/0x0178001C)

The SDCNTL register enables the DSP software to enable and disable the two banks of SDRAM individually. When an SDRAM bank is disabled, the two associated devices are put into a power-down mode, and the CE memory space is available for asynchronous expansion memory use. Table 1–32 summarizes the function of each bit in the SDCNTL register. Highlighted register values denote the power-up default values.

Table 1–32. DSP SDCNTL Register Bit Definitions

Bit	Name	Access	Description
7	–	–	–
6	–	–	–
5	–	–	–
4	–	–	–
3	–	–	–
2	–	–	–
1	CE3SDEN	RW	CE3 (bank 1) SDRAM enable (0 = disable, **1 = enable**)
0	CE2SDEN	RW	CE2 (bank 0) SDRAM enable (0 = disable, **1 = enable**)

1.8.7.9 DSP OSCB Register (0x01380020/0x01780020)

The OSCB register enables the DSP software to read an integer value that represents the frequency (MHz) of DSP oscillator B. The value is 0x32 (decimal 50) for the McEVM. Table 1–33 summarizes the function of each bit in the Oscillator B frequency register.

Table 1–33. DSP OSC B Register Bit Definitions

Bit	Name	Access	Description
7	FREQ7	R	Frequency bit 7 (0)
6	FREQ6	R	Frequency bit 6 (0)
5	FREQ5	R	Frequency bit 5 (1)
4	FREQ4	R	Frequency bit 4 (1)
3	FREQ3	R	Frequency bit 3 (0)
2	FREQ2	R	Frequency bit 2 (0)
1	FREQ1	R	Frequency bit 1 (1)
0	FREQ0	R	Frequency bit 0 (0)

1.8.7.10 DSP SEM0 Register (0x01380024/0x01780024)

The SEM0 register provides a single semaphore flag that can be used to share devices or coordinate activities between the host and the DSP. Only bit 0 (DSPSEM0) is valid in this register. A semaphore is requested by writing a ‘1’, to this bit. The bit is read back to determine if the semaphore was granted. If DSPSEM0 is read back as a ‘1’, then the DSP owns the semaphore. If it is read back as a ‘0’, then the DSP does not own the semaphore. It is important to note that a semaphore request must be explicitly made each time, since pending requests are not supported. When the host wants to relinquish the semaphore, a ‘0’ should be written to the DSPSEM0 bit. Table 1–34 summarizes the function of each bit in the SEM0 register.

The DSP accesses the same semaphore as the host, but only the host or the DSP can own it at any one time. At reset, the semaphore is set to ’0’ on both sides, so neither one owns the semaphore.

Table 1–34. DSP SEM0 Register Bit Definitions

Bit	Name	Access	Description
7		–	–
6		–	–
5		–	–
4		–	–
3		–	–
2		–	–

Bit	Name	Access	Description
1		–	–
0	DSPSEM0	RW	Semaphore 0 Flag (For reads: 0=not owned, 1=owned; For writes: 0=relinquish, 1=request)

1.8.7.11 DSP SEM1 Register (0x01380028/0x01780028)

The SEM1 register provides a single semaphore flag that can be used to share devices or coordinate activities between the host and the DSP. Only bit 0 (DSPSEM1) is valid in this register. A semaphore is requested by writing a ‘1’ to this bit. The bit is read back to determine if the semaphore was granted. If DSPSEM1 is read back a a ‘a‘1’, then the DSP does not own the semaphore. It is important to note that a semaphore request must be explicitly made each time, since pending requests are not supported. When the host wants to relinquish the semaphore, a ‘0’ should be written to the DSPSEM1 bit. Table 1–35 summarizes the function of each bit in the semaphore 1 register.

Table 1–35. DSP SEM1 Register Bit Definitions

Bit	Name	Access	Description
7		–	–
6		–	–
5		–	–
4		–	–
3		–	–
2		–	–
1		–	–
0	DSPSEM1	RW	Semaphore 1 Flag (For reads: 0=not owned, 1=owned; For writes: 0=relinquish, 1=request)

1.8.7.12 DSP FALC Control Register (0x0138002Cx0x0178002C)

The FALC control register controls the T1/E1 FALC and MVIP FMIC devices. Bits 7 through 4 of the DSP FALC control register are decoded to provide 16 timing modes (0-15) for the McEVM. Only four modes (0-3) are used for the current implementation. Modes 4-15 are equivalent to mode 0. The four modes define the states of the FALC_SYNC signal and FMIC_FRM_EN signal generated by the CPLD. The FALC_SYNC signal is connected to the SYNC pin of the FALC. The FMIC_FRM_EN signal controls the source of the frame signal connected to the EX_8KA pin of the FMIC. The four modes are defined in Table 1–36.

Table 1–36. Master Timing Select Modes

Mode	FALC_SYNC State	FMIC_FRM_EN State
0	Logic 0 selected	Logic 1 (daughterboard is timing master)
1	CLK2 from FMIC selected	Logic 1 (daughterboard is timing master)
2	Logic 0 selected	Logic 0 (FMIC is timing master)
3	CLK2 from FMIC selected	Logic 0 (FMIC is timing master)

The FMIC provides four local serial data streams (LD-0 to LD-3). In normal operation, LD-0 is connected to the DSP, LD-1 is connected to the daughterboard, LD-2 is connected to the FALC, and LD-3 is connected to the VBAP. Bit 3 of the FALC control register permits the connections for LD-3 and LD-1 to be swapped. Bit 2 of the FALC control register permits the connections for LD-2 and LD-0 to be swapped. Bit 1 and bit 0 of the FALC control register are used to generate reset signals to the FMIC and FALC respectively. Table 1–37 summarizes the function of each bit in the FALC control register. It is important to note that 0 is the default at reset or powerup for bits 4-7.

Table 1–37. DSP FALC Control Register Bit Definitions

Bit	Name	Access	Description
7	MSEL3	RW	Bit 3 of the mode select code
6	MSEL2	RW	Bit 2 of the mode select code
5	MSEL1	RW	Bit 1 of the mode select code
4	MSEL0	RW	Bit 0 of the mode select code
3	LD31SWP	RW	Swap the connections for LD-3 and LD-1 (**0 = normal operation**, 1 = swap 3/1)
2	LD20SWP	RW	Swap the connections for LD-2 and LD-0 (**0 = normal operation**, 1 = swap 2/0)
1	FMICRST	RW	Reset FMIC (**0= enable**, 1=reset)
0	FALCRST	RW	Reset FALC (**0=enable**, 1=reset)

1.8.7.13 DSP Interrupt Control Register (0x01380030/0x01780030)

The DSP interrupt control register provides access to raw and latched versions of interrupt signals generated by the McEVM daughterboard and the T1/E1 interface. Register bits 7 and 6 contain the state of the latched signals of the daughterboard and T1/E1 interface interrupts, respectively. Register bits 5 and 4 contain the state of the raw signals of the daughterboard and T1/E1 interface interrupts, respectively. Register bits 3 and 2 are not defined. Register bits 1 and 0 are used for clearing and masking the daughterboard and T1/E1 interface interrupts respectively. A ‘0’ value in a clear/mask bit enables the corresponding interrupt. A ‘1’ value in a clear/mask bit clears the corresponding interrupt. While a clear/mask bit contains a 1, the corresponding interrupt is masked. Table 1–38 summarizes the function of each bit in the interrupt control register.

Table 1–38. DSP Interrupt Control Register Bit Definitions

Bit	Name	Access	Description
7	LTDBINT	R	Latched daughterboard interrupt (0 = inactive, 1 = active)
6	LTFALCINT	R	Latched FALC interrupt (0 = inactive, 1 = active)
5	DBINT	R	Raw daughterboard interrupt (0 = inactive, 1 = active)
4	FALCINT	R	Raw FALC interrupt (0 = inactive, 1 = active)
3	–	–	–
2	–	–	–
1	CDBINT	RW	Clear/Mask daughterboard interrupt (**0 = enable**, 1 = clear/mask)
0	CFALCINT	RW	Clear/Mask FALC interrupt (**0 = enable**, 1 = clear/mask)

1.8.7.14 DSP Miscellaneous Status Register (0x01380034/0x01780034)

The miscellaneous status register bits 7 through 4 control user-defined LED indicators that are located on the mounting bracket. Bit 3 controls the selection of µ/A-Law PCM encoding, and bit 2 controls the VBAP enable/disable function. Bit 1 is a status bit used to report a FMIC error condition. Table 1–39 summarizes the function of each bit in the miscellaneous status register.

Table 1–39. DSP Miscellaneous Status Register Bit Definitions

Bit	Name	Access	Description
7	GRNLED1	RW	User-defined green LED #1 (**0=LED off**, 1=LED on)
6	GRNLED2	RW	User-defined green LED #2 (**0=LED off**, 1=LED on)
5	REDLED3	RW	User-defined red LED #3 (**0=LED off** , 1=LED on)
4	YELLED	RW	User-defined yellow LED (**0=LED off,** off, 1=LED on)
3	MU_ALAW	RW	μ/A-Law selection control (**0=**μ, 1=A)
2	VBAPEN	RW	VBAP enable (**0=disabled**, 1=enabled)
1	FMICERR	R	Raw FMIC error signal (0=no error, 1=error)
0	–	–	–

1.8.8 PCI and DSP Interrupt Control

The CPLD provides interrupt control that allows the host and DSP to interrupt each other and supports various McEVM interrupts related to the daughterboard, PCI controller, T1/E1 transceiver, and JTAG TBC. The CPLD also controls DSP interrupts that are used to drive PCI bus master transfers.

The host can interrupt the DSP with three different interrupts using the following methods:

- Setting the HPI control (HPIC) register DSPINT bit (DSPINT)
- Setting the PCI memory-mapped control register DSPNMI bit (NMI)
- Accessing a PCI controller mailbox (EXT_INT4)

The HPIC register can be accessed from the host by addressing offset 0 of PCI BAR3. A host write to this address, with the DSPINT bit set to 1, causes a DSPINT interrupt to the DSP. The CPLD provides a state machine that manages the data transfers between the PCI controller and the DSP HPI.

The CPLD provides a PCI memory-mapped board control register (CNTL) that can be accessed by the host at offset 0 of PCI BAR2. When the DSPNMI bit of this register is set to 1, it causes an NMI interrupt to the DSP if the host is selected by the DSP software as the NMI interrupt source.

The host can also interrupt the DSP by writing or reading a PCI controller mailbox register mapped into PCI BAR0. The incoming and outgoing mailbox registers can generate interrupts when they become full or empty. When the selected mailbox register conditions occur, the DSP’s EXT_INT4 interrupt is asserted. The CPLD logic inverts the S5933’s IRQ# active low output signal to generate a rising-edge EXT_INT4 signal because the DSP defaults to rising-edge interrupts.

The DSP can interrupt the host by setting the HPIC register’s HINT bit to 1. This action forces the DSP’s $\overline{\text{HINT}}$ output signal to be asserted low. The CPLD includes a falling-edge detector that generates a pulse on the S5933’s external mailbox clock (EMBCLK) input, which causes an interrupt on INTA# to the host. A second falling-edge detector is used to generate an interrupt to the host when the TBC interrupt is asserted. The host software can enable HPI and TBC interrupts via the PCI memory-mapped CNTL register.

Both the host and DSP software can poll interrupt signals by reading PCI and DSP memory-mapped registers in the CPLD. The host can poll the DSP host and TBC interrupts. The DSP can poll the daughterboard, host NMI, T1/E1 transceiver, and PCI interrupts. The ability to poll interrupt signals from the software, as well as using them to interrupt execution flow, provides flexibility to the programmer that may be useful in various applications.

The CPLD includes two state machines that drive PCI bus master read and write operations based on the status of the PCI controller's read and write FIFO flags. The PCI bus master state machines are individually enabled by the DSP software by setting bits in the DSP memory-mapped FIFOSTAT register implemented in the CPLD. When the PCIMREN bit is set, the master read state machine generates EXT_INT5 interrupts to the DSP when the PCI controller's read FIFO flag (RDEMPTY) indicates that the read FIFO is not empty. The DSP can respond to the external interrupt with background DMA transfers or an interrupt service routine to read data from the PCI controller. The state machine waits for a DSP EMIF read of the FIFO before repeating the process. The state machine must be disabled at the end of each PCI bus master read block transfer. The CPLD provides a similar state machine for PCI bus master writes. When the PCIMWEN bit is set, the master write state machine generates EXT_INT6 interrupts to the DSP when the PCI controller's write FIFO flag (WRFULL) indicates that the write FIFO is not full. Similar to read transfers, the DSP can use background DMA or an interrupt handler to write data to the PCI controller.

The DSP can be interrupted with four external maskable interrupts and one nonmaskable interrupt (NMI). All DSP interrupts on the 'C62x McEVM are asserted on rising edges. Table 1–40 summarizes the DSP interrupts on the McEVM.

Table 1–40. DSP Interrupts Usage

DSP Interrupt	Description of Use
EXT_INT4	PCI controller interrupts
EXT_INT5	PCI bus master reads DMA synchronization
EXT_INT6	PCI bus master writes DMA synchronization
EXT_INT7	T1/E1 transceiver and expansion peripheral interface (XEXT_INT7)
NMI	Host (PCI register)

EXT_INT4 indicates to the DSP that a significant event has occurred as a result of activity within the PCI controller. This activity may include mailbox full/ empty conditions, end of bus master transfers, a self-test request from the PCI bus, and transfer errors.

EXT_INT5 and EXT_INT6 are used to synchronize PCI bus master reads and writes, respectively. These interrupts are used to trigger the DSP's DMA controllers or a CPU interrupt service routine.

EXT_INT7 is provided to indicate an event associated with the T1/E1 transceiver has occurred. It is also shared with the expansion peripheral interface (as DB_INT) so that a daughterboard can interrupt the DSP to indicate significant events. The CPLD provides an interrupt controller that allows the daughterboard and T1/E1 transceiver to share the DSP's EXT_INT7 interrupt.

The NMI can be asserted by the host software via a memory-mapped control register on the PCI bus.

1.8.9 CE1 Memory Decoding/Data Transceivers Control

The CPLD decodes DSP EMIF CE1 memory accesses to control various McEVM hardware and other logic within the CPLD itself.

The McEVM uses ’ALVCH16245 data transceivers to provide data bus isolation and voltage translation. These transceivers are enabled by CPLD memory decode logic based on the CE1 memory accesses. DSP EMIF accesses to the PCI controller’s registers and FIFOs enable the PCI add-on bus transceiver. DSP EMIF accesses to the daughterboard enable the external data transceivers. DSP EMIF accesses to the MVIP FMIC, T1/E1 transceiver, handset audio codecs, CPLD registers, and daughterboard enable the global data bus transceiver. The direction of these transceivers is controlled by the DSP’s output enable ($\overline{\text{AOE}}$) signal, which defines the direction of the data access.

A CPLD state machine, along with combinatorial logic, is used to generate the asynchronous ready (ARDY) signal to the DSP. Because the McEVM includes different types of devices in the CE1 space, there are different timing requirements for each. The CPLD generates the ARDY signal based on the type of CE1 memory access. For PCI controller accesses, the ARDY signal is not asserted until the add-on bus in available, unless the board is operated standalone. In stand-alone operation, the ARDY signal is always asserted for PCI controller accesses so that the EMIF does not lock up waiting for ARDY. The external ready from the daughterboard is presented to the DSP when expansion memory is accessed. The CPLD provides the ready logic that extends the asynchronous strobe time as needed for each particular access.

The CE2 and CE3 DSP memory spaces default to two banks of 4M × 32-bit SDRAM. However, if one or both of these banks of SDRAM are not required for an application, and more or faster asynchronous memory or memory-mapped devices are required, CE2 and CE3 can support expansion memory on a daughterboard. The CE3SDEN and CE2SDEN bits in the CPLD’s SDCNRL register determine whether the SDRAM banks are enabled or disabled. If a bank is disabled, it is put into a low-power mode and does not respond to accesses. Additionally, CPLD logic enables external accesses to asynchronous memory in these spaces by activating the expansion memory data transceivers.

1.8.10 User Options Control

The McEVM provides user options to select the DSP’s input clock source, endian and boot modes, and the JTAG emulation method. The CPLD includes user options control logic that selects between hardware DIP switch and software switch user options. Table 1–41 summarizes the McEVM user options.

Table 1–41. User Options Summary

User Option	Description	Number of Signals
Boot mode	Selects no-boot, HPI-boot, or ROM-boot	5
Clock mode	Selects multiply-by-1 (no PLL) or multiply-by-4 (PLL) clock mode	1
Clock select	Selects 33.25 MHz or 50 MHz for CLKIN	1
Endian select	Selects big- or little-endian memory addressing	1
JTAG select	Selects internal or external JTAG emulation	1
User-defined	User-defined options	3

The boot mode option selects how the DSP boots upon the release of its reset input signal. It can begin execution immediately with no boot, or it can be booted from ROM in the CE1 space or from the host port interface (HPI). The five control signals select the type of boot, the type of memory located at address 0, and the memory map (MAP 0 or MAP 1) that is to be used.

The clock mode selects whether the CPU clock (CLKOUT1) is the same as CLKIN (multiply-by-1) or four times CLKIN (multiply-by-4). The ’C6201 provides two pins to select the clock mode. However, both pins have the same value in each of these modes, so only one control signal needs to be used. The clock mode pins are both set to 0 for multiply-by-1 mode and 1 for multiply-by-4 mode.

The clock selection determines which of the two onboard clock sources is used. The McEVM provides both 33.25- and 50-MHz clocks. If multiply-by-4 clock mode is selected, this results in CPU clock rates of 133 MHz and 200 MHz, respectively.

The endian selection determines whether the DSP uses little- or big-endian byte/halfword addressing.

The JTAG selection determines whether the onboard JTAG controller or an external XDS510 emulator is to be used for debugging. When the ’C62x McEVM operates outside the PC in stand-alone mode, the JTAG selection is forced by the CPLD to external XDS510 use.

Three user-defined options are provided for application use. These user options can be read by both the host and the DSP software via CPLD registers.

Because the McEVM can operate in both PCI and stand-alone environments, it must support the selection of user options in both situations. When the

McEVM is operated outside the PC, DIP switches are used to configure the board. When the McEVM is in the PC, configuration is done via software switches under software control to eliminate the need to remove the PC's cover. This dual-use option support is transparent because it defaults to the standard DIP switch control but allows for software switch override, if desired. If software does not control the configuration from the host side, the McEVM defaults to the DIP switch settings. Therefore, in external operation, the DIP switches are used exclusively for user option selections.

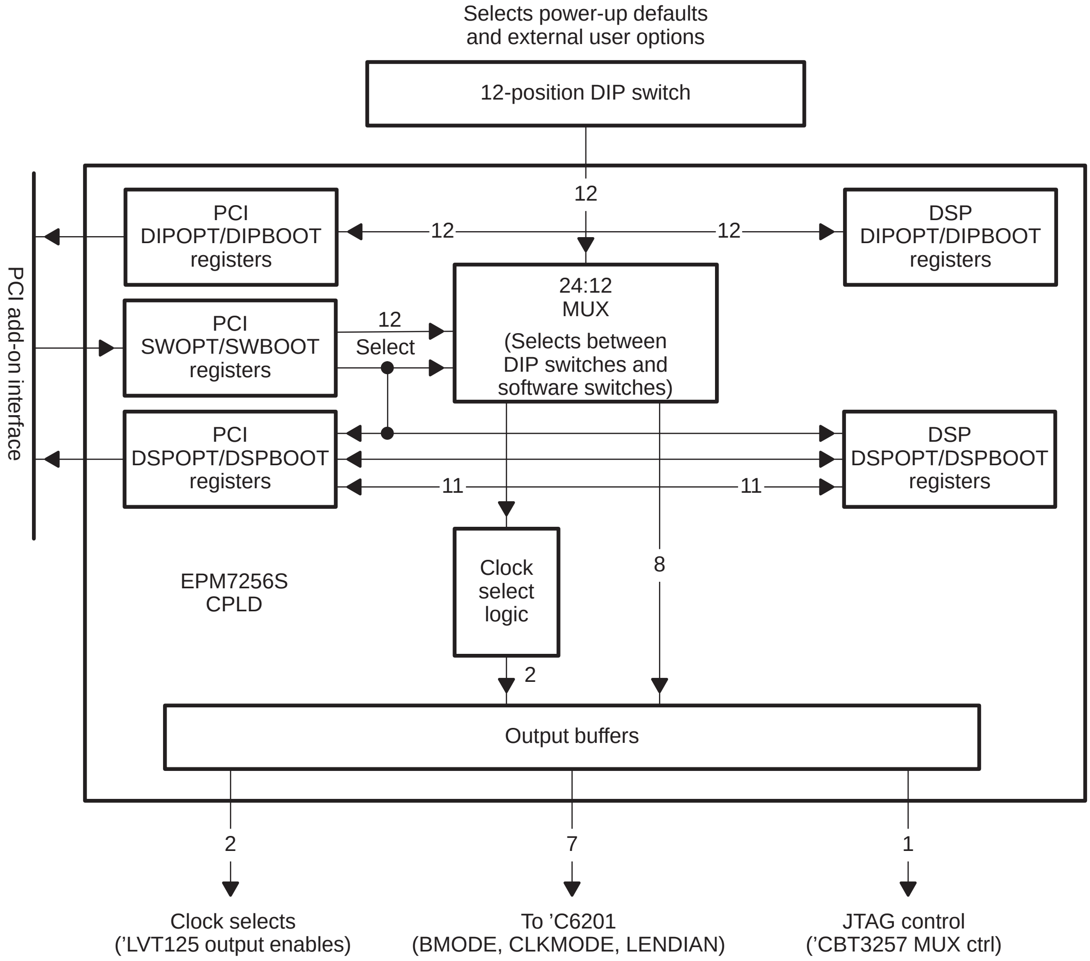
Figure 1–11 shows how this dual-use option support is implemented in the McEVM's CPLD.

The McEVM's CPLD includes a multiplexer that selects between the hardware DIP switches and PCI-controlled software switches. Upon power up, the CPLD defaults the configuration to the DIP switch settings, providing transparent operation. Memory-mapped registers on the PCI bus allow the host McEVM driver to configure the board and DSP directly to override the hardware DIP switch settings, if desired.

Memory-mapped registers in the CPLD allow both the host and DSP software to observe DIP switches and the current DSP option selections. When the McEVM is under software switch control, the 12 DIP switches can be used by the DSP software for other purposes.

Based on the selected configuration, the CPLD provides control signals to the DSP and external hardware for clock and JTAG selection. Transparent DIP switch configuration is provided upon power up, providing power-up defaults without the need for software control.

Figure 1–11. Dual-Use Option Support


Selects power-up defaults
and external user options
12-position DIP switch
12
PCI DIPOPT/DIPBOOT registers
12
12
DSP DIPOPT/DIPBOOT registers
PCI add-on interface
PCI SWOPT/SWBOOT registers
12
Select
24:12 MUX (Selects between DIP switches and software switches)
PCI DSPOPT/DSPBOOT registers
DSP DSPOPT/DSPBOOT registers
11
11
EPM7256S CPLD
Clock select logic
8
2
Output buffers
2
7
1
Clock selects ('LVT125 output enables)
To 'C6201 (BMODE, CLKMODE, LENDIAN)
JTAG control ('CBT3257 MUX ctrl)

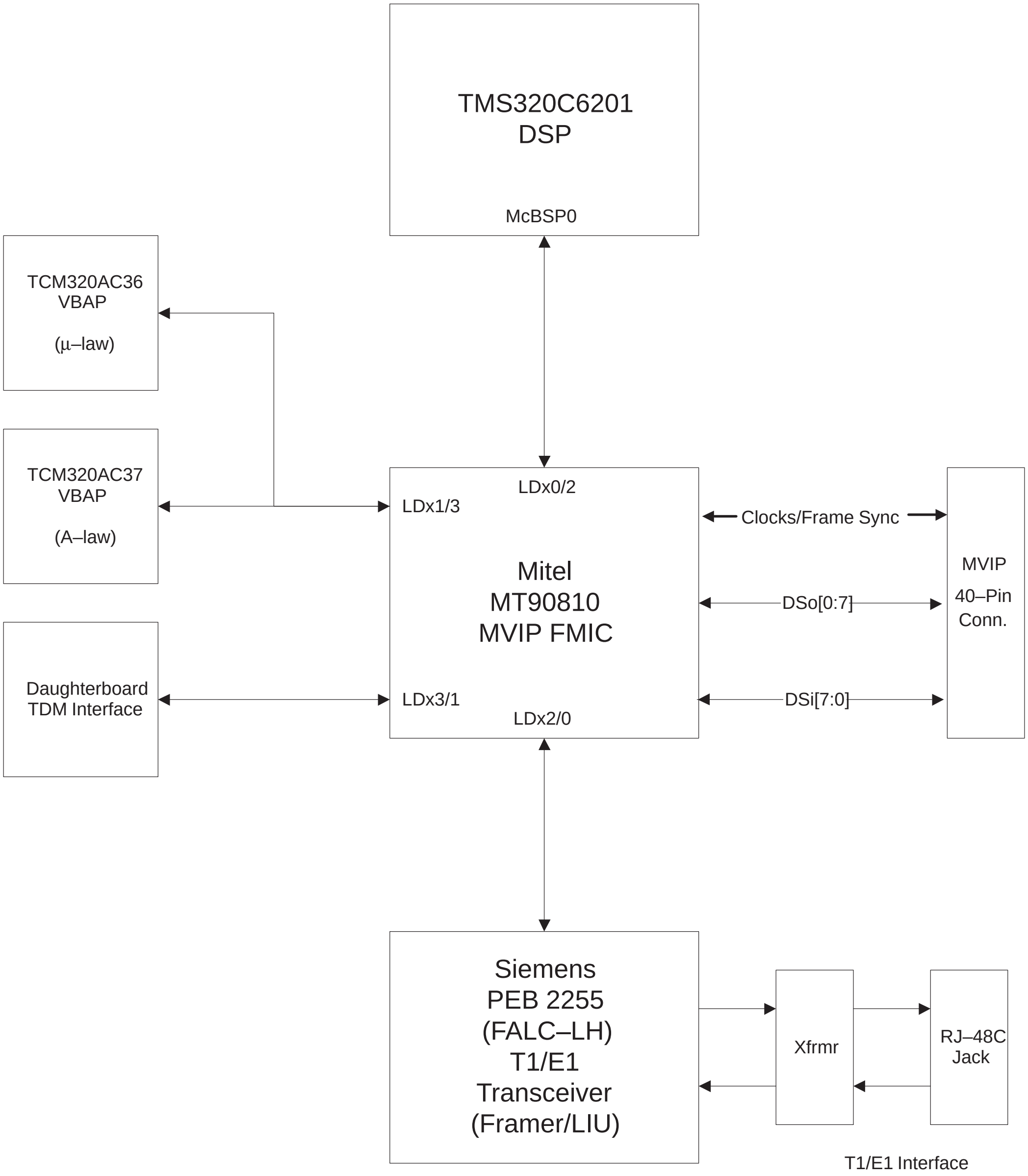
1.9 MVIP Interface and Switch

The McEVM includes a fully compliant, enhanced MVIP-90 interface based on the Mitel MT90810 Flexible MVIP Interface Circuit (FMIC). The MVIP interface allows the McEVM to interoperate with a large base of telephone interface resources such as trunk interfaces, voice, video, fax, text-to-speech, and speech recognition boards. The MVIP interface could also be used to interconnect multiple McEVM boards via the MVIP TDM bus if desired. The TDM bus can be accessed through a connector provided at the top of the McEVM board.

In addition to meeting the requirements of the MVIP interface, the FMIC provides a switch that provides 384x384 channel non-blocking connectivity between all telephony devices on the board and the DSP via its McBSP0 serial port. This design allows the DSP to have access to every MVIP, T1/E1 and companded handset audio time slot on the board, resulting in optimal utilization of the DSP's serial port and providing a very flexible switching arrangement. Additionally, audio monitoring and testing of various telephone channels can be accomplished by transferring samples to and from the handset interface via the FMIC.

The default MVIP configuration uses all four local streams operating with 32 timeslots at a rate of 2.048 Mbps each. For some applications, other configurations may be useful, such as when you may want to bring 64 or 128 timeslots to the DSP. The FMIC supports different local stream modes of operation which result in different local streams being active. The McEVM design provides DSP-controlled local stream switching (via the CPLD's FALC control register) to support this type of configuration. Local stream multiplexers on the McEVM allow local streams 1 and 3, as well as 0 and 2 to be swapped in order to support higher throughput FMIC modes of operation. Figure 1–12 shows the MVIP FMIC interfaces. Note that the default local data stream assignments are shown first, with the optional swapped local data streams indicated second.

Figure 1–12. McEVM's MVIP interface



The FMIC supports eight pairs of 2.048 Mbit/sec data streams from the MVIP telephony bus, along with four local pairs of 2.048 Mbit/sec data streams. All data streams are synchronized to the same master clock and 8 kHz frame sync signals. Four local serial interfaces connect the FMIC to the following components:

- ’C6201 DSP
- VBAP handset interface
- T1/E1 transceiver
- A spare time-division multiplexed (TDM) port brought out to the expansion peripheral connector

All data streams consist of 8-bit data (typically companded with μ-law or A-law) time slots sent at an 8-kHz rate, resulting in 64-Kbps time slots.

One of the MVIP FMIC’s 2.048-Mbps local serial interfaces is connected to the ’C6201 McBSP0 interface through a pair of TI ’CBT3257 quad 2:1 multiplexers as shown in Figure 1–13. The ’CBT devices perform voltage translation between the 3.3-V DSP and the 5-V FMIC and allow the McBSP0 serial port to be connected to either the MVIP FMIC or a daughterboard at any one time, depending on your application. This allows a daughterboard to use both of the DSP’s serial ports, which is very important for many applications such as digital subscriber line technologies (xDSL). The McEVM defaults to connection to the daughterboard, and the FMIC’s initialization library function controls a memory-mapped register bit in the CPLD to connect the FMIC to the DSP’s McBSP0 interface.

The FMIC, which is mapped into the DSP’s asynchronous CE1 memory space, provides an 8-bit microprocessor interface that is compatible with the ’C6201 EMIF timing when configured for Intel non-multiplexed bus timing. The FMIC presents four read/write, 8-bit registers to the DSP which consist of master control/status, low address, address mode, and indirect data registers. The FMIC uses indirect addressing to allow the DSP to access other control registers, data memory, and connection memory. The DSP interfaces to the 5-V FMIC via TI ’ALVCH buffers and transceivers to perform voltage translation.

Table 1–42. MVIP FMIC Registers

DSP Address MAP 1 (MAP 0)	MVIP FMIC Register Description	Access
01740000 (01340000)	Master control/status register (MCR)	Read/write
01740004 (01340004)	Low address register (LAR)	Read/write
01740008 (01340008)	Address mode register (AMR)	Read/write
0174000C (01340008)	Indirect data register (IDR)	Read/write

The DSP can configure the FMIC’s connection memory to select which time slots are routed to other time slots. For example, the DSP can configure the

FMIC to send it only T1/E1 time slots or, as another example, some MVIP time slots along with the handset audio time slots. The DSP has complete control over where all time slots are sent.

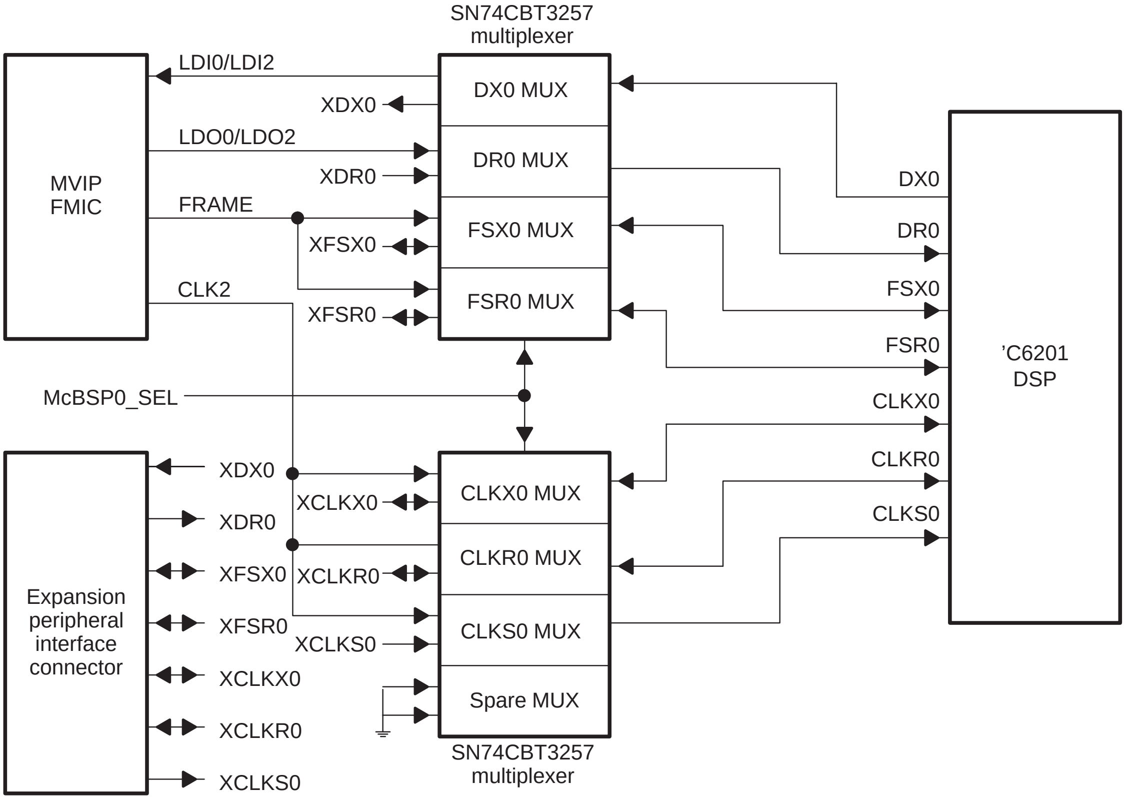
The FMIC supports all of the MVIP clocking modes, allowing the McEVM telephony data streams to be synchronized by the MVIP master clock, a local crystal, or an external 8-kHz clock that an on-chip PLL uses to generate a 4.096-MHz clock. The DSP controls which clock source is used as the master clock.

The MVIP interface consists of a 40-pin, right-angle connector at the top of the McEVM board. See Appendix A for the connector pinout.

Complete information on the MVIP FMIC device, including register definitions, is available at the following URL:

http://www.mitelsemi.com/products/pdf/mt90810.pdf

Figure 1–13. McBSP0 Selection



1.10 T1/E1 Interface

The McEVM provides a common T1/E1 interface that allows it to connect to T1, E1, or Integrated Service Digital Network (ISDN) primary rate trunks operating at 1.544 Mbps or 2.048 Mbps. This T1/E1 interface provides a multichannel, digital telephone interface for the ’C6201 DSP that is ideal for processing multiple channels.

WARNING

This interface is electrically compatible with T1, E1, and ISDN services provided by the phone company. It is NOT certified or approved for direct connections.

The T1/E1 interface is based on the Siemens FALC-LH (PEB2255) long-haul, fully-integrated T1/E1 framer and line interface unit (LIU). The FALC-LH is a 5-V device which combines a sophisticated framer, transmit/receive slip buffers, and a physical line interface into one device that supports both T1 and E1. This eliminates the need to have two framers and associated interfaces on the McEVM, resulting in a single framer/driver and minimal board space.

The FALC-LH provides a 2.048 Mbps serial data stream that is synchronized to the board’s master clock. This is the common rate that is used for all interfaces, therefore the FALC-LH supports this system bus clock rate even when 1.544 Mbps of T1 data is being received. The serial data stream is routed to the FMIC for switching to the desired output stream time slot, which could be the handset, the DSP or the MVIP bus.

The DSP interfaces to the FALC-LH via an 8/16-bit microprocessor interface that allows the DSP to control and receive status from the device. The FALC-LH’s interface is compatible with the ‘C6201 EMIF asynchronous timing when it is operating in the asynchronous Motorola (demultiplexed address/data) mode. The FALC-LH is mapped into the DSP’s CE1 external memory space. The FALC-LH provides many registers and features that can be controlled by the DSP, such as alarm/error monitoring and signaling supervision. The DSP can also put the device in several different types of loopback modes (both analog and digital) for diagnostics, maintenance and troubleshooting. Other extensive test and diagnostic functions, such as PRBS test pattern generation are also available.

Another feature of FALC-LH is that it provides an interrupt output that is used by a CPLD interrupt controller to notify the DSP of significant framer events using the EXT_INT7 interrupt.

The FALC-LH's physical line interface circuit recovers clock and data from analog signals with +3 to -43 dB cable attenuation, appropriate for both short (-18 dB) and long haul T1/E1 applications. Receive line equalization is provided and programmable transmit pulse shaping, using a minimum number of external components, is provided. Data and clock jitter attenuation can be inserted on either the receive or transmit paths. A complementary driver output is provided to couple 75/100/120 ohm lines via an external transformer.

The FALC-LH analog outputs are routed to a dual-transformer for the physical interface to the T1 or E1 short-haul line.

The T1/E1 output is presented to the user as an RJ-48C twisted-pair modular jack located on the mounting bracket. The jack's wiring configuration is provided in Appendix A.

Complete information on the FALC-LH T1/E1 device, including register definitions, is available at the following URL:

http://www.siemens.de/semiconductor/products/ics/33/falc_lh.htm

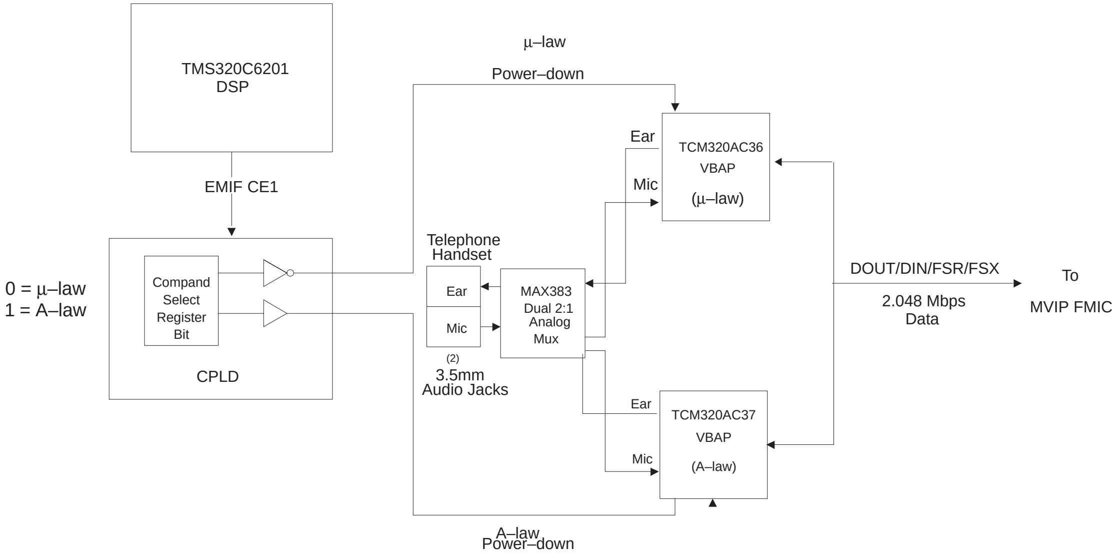
1.11 Handset Interface

The McEVM includes a handset interface that allows the DSP to process voice data or route it to the TI/E1 and MVIP interfaces. Likewise, T1/E1 and MVIP data can be sent directly to the handset earphone for monitoring with or without requiring DSP intervention.

The telephone handset interface supports both A-Law and µ-Law PCM companding. Two TI voice-band audio processor (VBAP) devices, which are synchronized to the master telephony clock and framing signals, are used to provide A-Law and µ-Law PCM companding. Only one companding format is selected and used at a time. The handset interface defaults at power up to µ-Law companding as a result of the default reset value of a register bit in the CPLD.

The DSP's software directly controls the interface with a single memory-mapped register bit in the CPLD. The value of the register bit results in two, mutually exclusive control bits that are routed to the power-down ($\overline{\text{PDN}}$) pins of both VBAP devices. When one VBAP is enabled, the other one is in power-down mode with its output amplifier disabled and its digital output tristated. This design allows the two VBAP devices to be connected to support both companding formats, appearing just like a single dual-mode device to the DSP's software. Figure 1–14 shows the configuration of the VBAP devices.

Figure 1–14. McEVM VBAP Configuration



The TI TCM320AC36 device provides the μ-Law companding, and the TCM320AC37 device provides the A-Law companding. The VBAPs are single-supply devices with glueless interfaces to an electret microphone, an earphone, and the MVIP switch. Only a few passive components (resistors and capacitors) are required for biasing, gain, and DC blocking. Because the VBAPs are interfaced to the 5-V MVIP FMIC switch and not directly to the ’C6201, no voltage translation is required, allowing a direct connection. Additionally, because one of the devices is always guaranteed to be in power-down mode by the board’s CPLD logic, forcing its output to be tristated, the two devices’ output data to the MVIP switch can be connected. This design eliminates the need for a strap or switch option and saves a local serial port on the MVIP switch that is brought out to the peripheral expansion connector, for external use. An analog multiplexer is controlled by one of the VBAP control bits to connect the microphone and earphone to the selected VBAP device.

The VBAPs can provide 13-bit linear outputs; however, for the telecom subsystem, they are configured for 8-bit companding to be directly compatible with the T1/E1 framer and MVIP data formats. The ’C6201 supports both A-Law and μ-Law PCM formats directly and converts them to linear format in hardware for processing. An added benefit of the companded data formats is that data can also be routed directly to and from the MVIP and the T1/E1 framer to support a wide variety of applications, such as monitoring channels and dropping and inserting voice and test signals.

The telephone handset interface is brought out to two 3.5mm audio jacks on the McEVM’s mounting bracket. One audio jack provides a mono microphone input and the other provides a mono earphone output. Appendix A provides the pinout of the audio jacks.

1.12 Power Supplies

The McEVM requires 1.8-V, 3.3-V, and 5-V supplies. An optional daughterboard –12-V supply is brought to the peripheral expansion interface from both the PCI bus and the external power connector. The 5-V is obtained from the PCI bus during internal operation or an external power connector during external operation. The McEVM's PCI connector's PRSNT1# and PRSNT2# pins are configured to indicate a maximum of 25-watts power dissipation. The 1.8-V and 3.3-V voltages for McEVM are provided by onboard switching regulators that use the PCI or external 5-V supply as its input.

1.12.1 3.3-V Voltage Regulator

The DSP's I/O buffers, SBSRAM, SDRAM, low-voltage buffers/transceivers, and CPLD I/O buffers require 3.3 V. The 3.3-V supply is provided by an integrated switching regulator (ISR) (part number PT6405B). The PT6405B provides 3.3 V at up to 3 A. The device's output voltage defaults to 3.3 V, but it can be adjusted in the range of 2.8 V–3.8 V by changing external resistors. The McEVM layout supports the ability to adjust the voltage with resistors, but these are not installed during manufacturing. This surface-mount device is an efficient (85%) switching regulator that is mounted flat on the board with a 0.38-inch height that is compatible with a PCI slot. This regulator is positioned on the board so that it does not interfere with the daughterboard interface.

The PT6405B only requires a couple of external capacitors, so its external interface is similar to a linear regulator because it is a module. It therefore provides the advantages of a switching regulator (efficient and runs cool) in a package that is easy to use, requiring no specialized layout or multiple devices that are required by a custom switcher design. It has already been carefully designed for minimal emissions and has been tested.

1.12.2 1.8-V Voltage Regulator

The PT6407E regulator provides the DSP's 1.8 V core voltage with up to 3A of current. Large dynamic current requirements of the DSP are supported by the 100-μ F external capacitor at the voltage regulator's output. In addition to a carefully designed bulk and decoupling capacitor arrangement.

1.12.3 External Power Connector

The 'C62x McEVM's external power connector is an industry-standard Molex 4-pin connector that is commonly used in PCs for disk-drive power. The external power connector is located at the bottom of the board with its pins oriented downward to prevent connection while the board is installed in the PC. When the McEVM operates outside the PC, the power connection is easily made at the edge of the board.

The four pins on the connector are used for 5 V, –12 V, GND, and 12 V. The standard disk-drive power cables do not supply –12 V, so if this is required by a daughterboard, an appropriate power supply should be used. If –12 V is not used on the daughterboard, then a standard disk-driver power cable can be used.

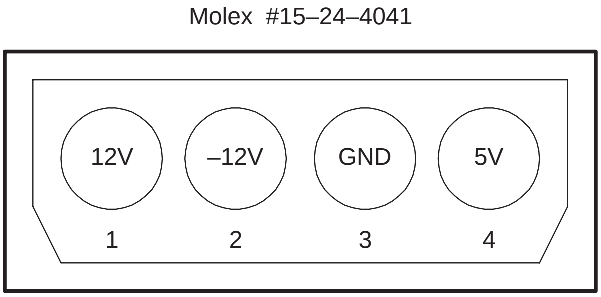
The user-supplied, external power supply should provide the following voltages and currents:

- ❑ 5 V_{DC} at 4 A
- ❑ 12 V_{DC} at 500 mA
- ❑ –12 V_{DC} at 100 mA (if required by a daughterboard)

These are the voltages and their respective currents available to the ’C62x McEVM in a PCI slot.

Figure 1–15 shows the orientation of the external power connector pins.

Figure 1–15. External Power Connector



Note: Drawing is not to scale.

Preventing Power Supply Damage

To prevent power supply damage, DO NOT connect power to the external power connector on the McEVM board when it is installed in a PCI slot on your computer. The external power connector is only for use in standalone operation.

1.12.4 Fan Power Connector

The ’C62x McEVM includes a connector to supply 5 V (at 100 mA) for an optional fan. The heat sink/fan is not required on TMS320C6201B.

1.13 Voltage Supervision

The McEVM’s dual-voltage supervision and reset generation are provided by a single voltage supervisor device. The voltage supervisor monitors the 3.3-V and 1.8-V power supplies, provides manual reset switch debounce, and generates a clean, 140-ms minimum reset pulse.

Because the 1.8–V and 3.3-V power supplies are derived from the 5-V power supply, they reach their values after the 5 V, and their status inherently indicates the status of the 5 V. The board is held in reset until the two voltages are within specification. Whenever the 3.3-V supply is below 3 V, including during power up, the device forces a reset until the threshold is met. The voltage supervisor supports the monitoring of a secondary voltage on its power failure input. A threshold that causes reset to occur if there is a drop in the 1.8 V, is set via two external resistors configured as a voltage divider.

The device also has a manual switch reset input that allows you to manually reset the DSP. This is important, particularly for the external McEVM operation. The McEVM’s CPLD asserts this manual reset input whenever you press the manual reset pushbutton or a software reset is asserted by the PCI controller.

The voltage supervisor has two reset outputs: one is active low and the other is active high. The active low reset is used for CPLD initialization and board reset control. The active high reset is used directly to control the SBSRAM ZZ (snooze) input, which puts the SBSRAM in a power-down mode during reset. This capability is part of the power management features that are included on the ’C62x McEVM.

1.14 User Options

The ’C62x McEVM includes a 12-position DIP switch (SW2-1 through SW2-12) that allows selection of user options. The user options include the DSP’s boot and clock modes, clock oscillator selection, endian mode, JTAG selection, and three user-defined options.

1.14.1 DIP Switches

Table 1–43 summarizes the 12 switches and their functions in both the ON and OFF settings. Table 1–44 lists the valid boot mode selections for the McEVM. Highlighted table entries indicate the default switch settings. If a daughterboard that provides boot memory is used, you must select the appropriate ROM boot mode from the *TMS320C6201/C6701 Peripherals Reference Guide*. An ON setting translates to a logic 0, and an OFF setting translates to a logic 1.

Table 1–43. User Option DIP Switches

Switch Number	Name	OFF Selection	ON Selection
SW2-1– SW2-5	BOOTMODE4– BOOTMODE0	See Table 1–44	See Table 1–44
SW2-6	CLKMODE	Multiply-by-1 mode (PLL bypassed)	**Multiply-by-4 mode**
SW2-7	CLKSEL	OSC B (50 MHz)	**OSC A (33.25 MHz)**
SW2-8	ENDIAN	Big endian	**Little endian**
SW2-9	JTAGSEL†	Internal (TBC)	**External (XDS510)**
SW2-10	USER2	1	**0**
SW2-11	USER1	1	**0**
SW2-12	USER0	1	**0**

† The JTAGSEL selection is ignored in stand-alone mode where external operation is forced.

Table 1–44. Valid Boot Mode Selections

Boot Mode	Map	Memory at Address 0	BM4 (SW2-1)	BM3 (SW2-2)	BM2 (SW2-3)	BM1 (SW2-4)	BM0 (SW2-5)
None	MAP 0	1/2-rate SBSRAM	ON	ON	ON	OFF	OFF
None	MAP 0	1×-rate SBSRAM	ON	ON	OFF	ON	ON
None	MAP 1	Internal	ON	ON	OFF	ON	OFF
HPI	MAP 0	External	ON	ON	OFF	OFF	ON
HPI	MAP 1	Internal	**ON**	**ON**	**OFF**	**OFF**	**OFF**

1.14.2 Jumper Options

Two jumper options (J5 and J6) on the McEVM are related to the MVIP interface.

The MVIP 2.048 MHz (C2o) and 4.096 (C4b) clocks can be terminated using a series 1000 pF capacitor and 1 Kohm resistor. J5 is used for C2o termination and J6 is used for C4b termination. Termination is selected by connecting pins 1 and 2 on these 3-pin headers. The manufacturing default is to connect pins 2 and 3 (no termination). Termination is only required if the McEVM is on the end of the MVIP bus.

Table 1–45 summarizes the jumper options. Note that all default settings are to connect pins 2 and 3 as shown in boldface.

Table 1–45. Jumper Options

Jumper Selection	Jumper Setting
MVIP C2o terminated	J5-1 to J5-2
MVIP C2o not terminated	**J5-2 to J5-3**
MVIP C4b terminated	J6-1 to J6-2
MVIP C4b not terminated	**J6-2 to J6-3**

1.15 Indicators

The McEVM has seven LEDs that provide a power-on indication, two user-defined status indications, and four T1/E1 status indicators.

A green LED is hardwired on the 5-V supply, which can originate from the PCI bus during internal operation or the power connector during external operation. When the green LED is illuminated, it indicates that power is applied to the board.

Two red LEDs are controlled by CPLD register bits that are memory mapped in the DSP's CE1 memory space. Both LEDs are located on top of the board. They are also under user DSP control and are application dependent.

The T1/E1 status indicators include red alarm, yellow alarm, sync, and loop indicators. A red LED is used for the red alarm indication, a yellow LED is used for the yellow alarm indication and the two green LEDs are used for the sync and loop conditions. The red alarm LED will be illuminated when a red alarm condition is declared for the received T1/E1 signal. The yellow alarm LED will be illuminated when a yellow alarm signal is received from the far-end terminal of the T1/E1 span. Because the LEDs are memory mapped in the DSP memory space, the host software can also control them from the HPI via the PCI bus. Table 1–46 provides a summary of the LED indicators.

Table 1–46. LED Summary Table

Bracket Location	Name	Color	Intended Indication
Bottom Left	GRN1	Green	Sync (Green)
Top Left	GRN2	Green	Loop (Green)
Top Right	RED3	Red	Red Alarm
Bottom Right	YEL	Yellow	Yellow Alarm

Chapter 2

TMS320C62x McEVM Host Support Software

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

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

- Reset and configure the 'C62x
- Load and execute code
- Send and receive messages
- Send and receive data streams
- Access board resources via the HPI

You can use the DLL functions described in section 2.3.2, *McEVM Win32 DLL API Functions*, to perform all of these operations.

Topic **Page**

2.1 Host Support Software Components

The host support 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 McEVM 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 McEVM software installation.

2.2 McEVM 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 McEVM boards in a system. For Windows 95, a Windows VxD, evm6x.vxd, is the low-level driver that provides access to the McEVM hardware. For Windows NT 4.0, a kernel mode driver, evm6x.sys, provides the McEVM hardware access.

All the functionality required for control and communication with the McEVM hardware is provided by a Win32 DLL, evm6x.dll, which handles the details of the low-level driver access. This DLL presents a common Win32 API for both Windows 95 user applications and Windows NT 4.0 user applications. Thus, a Win32 user mode application using the Win32 DLL executes under either operating system.

2.3 McEVM Win32 DLL API

This section describes the McEVM Win32 DLL data types and provides summaries, in alphabetical order, of the McEVM Win32 DLL API functions.

2.3.1 McEVM Win32 DLL API Data Types

Example 2–1 provides definitions for the Win32 DLL API data types.

Example 2–1. Win32 DLL API Data Types

(a) Board types

```
typedef enum {
   TYPE_UNKNOWN = 0,
   TYPE_EVM,
   TYPE_MCEVM
} EVM6XDLL_BOARD_TYPE, *PEVM6XDLL_BOARD_TYPE;
```

The EVM6XDLL_BOARD_TYPE type definition defines the possible values returned by the evm6x_board_type() function. A properly functioning McEVM board returns a board type of TYPE_MCEVM.

(b) Boot modes

```
typedef enum {
   NO_BOOT = 0,
   HPI_BOOT,
   ROM8_BOOT,
   ROM16_BOOT,
   ROM32_BOOT,
   NO_BOOT_MAP0,
   HPI_BOOT_MAP0,
   ROM8_BOOT_MAP0,
   ROM16_BOOT_MAP0,
   ROM32_BOOT_MAP0
} EVM6XDLL_BOOT_MODE, *PEVM6XDLL_BOOT_MODE;
```

The EVM6XDLL_BOOT_MODE type definition defines the valid values passed to the evm6x_reset_dsp() function. These values support all of the valid boot modes of the McEVM board. The enumerations that do not explicitly describe a ’C62x memory map use the MAP 1 memory map.

Example 2–1. Win32 DLL API Data Types (Continued)

(c) Clock modes

```
typedef enum {
    DSP_CLOCK_NORMAL = 0,
    DSP_CLOCK_SBSRAM,
    DSP_CLOCK_BX1,
    DSP_CLOCK_AX1
} EVM6XDLL_CLOCK_MODE, *PEVM6XDLL_CLOCK_MODE;
```

The EVM6XDLL_CLOCK_MODE type definition defines the valid values passed to the evm6x_set_board_config() function to set the board's clock configuration.

(d) Endian configuration

```
typedef enum {
    LITTLE_ENDIAN_MODE = 0,
    BIG_ENDIAN_MODE
} EVM6XDLL_ENDIAN_MODE, *PEVM6XDLL_ENDIAN_MODE;
```

The EVM6XDLL_ENDIAN_MODE type definition defines the valid values passed to the evm6x_set_board_config() function to set the board's endian configuration.

(e) Send and retrieve messages

```
typedef ULONG EVM6XDLL_MESSAGE, *PEVM6XDLL_MESSAGE;
```

The EVM6XDLL_MESSAGE type definition defines the type used as the message value passed to and from the McEVM board. This type is used in the evm6x_send_message() and the evm6x_retrieve_message() functions.

2.3.2 McEVM Win32 DLL API Functions

The following alphabetical listing includes all of the Win32 DLL API functions. Use this listing as a table of contents to the Win32 DLL API functions.

Function	Description	Page
evm6x_abort_read	Terminate the current read request	2-7
evm6x_abort_write	Terminate the current write request	2-8
evm6x_board_type	Return board type	2-9
evm6x_clear_message_event	Clear the incoming message event	2-10
evm6x_close	Close a driver connection to a board	2-11
evm6x_coff_display	Display COFF section information	2-12
evm6x_coff_load	Load a COFF image into DSP memory	2-13
evm6x_cpld_read_all	Read the contents of the CPLD registers	2-15
evm6x_dll_revision	Read EVM DLL Revision Information	2-16
evm6x_generate_nmi_int	Generate an NMI to a DSP	2-17
evm6x_hpi_close	Close the HPI for a DSP	2-18
evm6x_hpi_fill	Fill DSP memory using the HPI	2-19
evm6x_hpi_generate_int	Interrupt a DSP using the HPI	2-21
evm6x_hpi_open	Open the HPI for a DSP	2-22
evm6x_hpi_read	Read DSP memory using the HPI	2-23
evm6x_hpi_write	Write DSP memory using the HPI	2-27
evm6x_init_emif	Initialize the EMIF registers	2-31
evm6x_mailbox_read	Read from an EVM mailbox	2-32
evm6x_mailbox_write	Write to an EVM mailbox	2-33
evm6x_nvram_read	Read NVRAM memory on a board	2-33
evm6x_nvram_write	Write NVRAM memory on a board	2-35
evm6x_open	Open a driver connection to a board	2-36
evm6x_read	Read data from a boards PCI interface	2-37
evm6x_read_single	Read a single byte, short or long, using the HPI	2-25
evm6x_reset_board	Completely reset a board	2-39
evm6x_reset_dsp	Reset the DSP on a board and set boot mode	2-40
evm6x_retrieve_message	Get a message sent by a DSP	2-41

Function	Description	Page
evm6x_send_message	Send a message to a DSP	2-43
evm6x_set_board_config	Set the board configuration settings	2-44
evm6x_set_timeout	Set the data transfer time out value	2-46
evm6x_unreset_dsp	Release the DSP from its reset state	2-47
evm6x_user_semaphore_get	Acquire user semaphore	2-48
evm6x_user_semaphore_release	Release user semaphore	2-48
evm6x_user_semaphore_wait	Wait until semaphore available	2-49
evm6x_write	Write data to a boards PCI interface	2-49
evm6x_hpi_write_single	Write a single byte, short or long, using the HPI	2-29

evm6x_abort_ read

Terminates a Pending Read Transfer

Syntax

#include <evm6xdll.h>
BOOL **evm6x_abort_read(**HANDLE *h_device***);**

Description

The evm6x_abort_read() function terminates a pending read operation for a target board. This can be used by one thread to terminate the pending read operation of another thread.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_abort_read() function aborts a read operation that has not finished. The evm6x_read call is pending in a separate thread.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* Start thread to receive data */
. . .
    /* Do something else, then check that receive is */
    /* complete                                      */
. . .
    /* Abort read operation that is not complete */
        if ( !evm6x_abort_read( h_board ) )
        {
            /* evm6x_abort_read() failed */
        }
```

evm6x_abort_write

Terminates a Pending Write Transfer

Syntax

#include <evm6xdll.h>
BOOL **evm6x_abort_write(**HANDLE *h_device***);**

Description

The evm6x_abort_write() function terminates a pending write operation for a target board. This can be used by one thread to terminate the pending write operation of another thread.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_abort_write() function aborts a write operation that has not finished. The evm6x_write call is pending in a separate thread.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* Start thread to send data */
. . .
    /* Do something else, then check that transfer is */
    /* complete                                      */
. . .
    /* Abort write operation that is not complete */
        if ( !evm6x_abort_write( h_board ) )
        {
            /* evm6x_abort_write() failed */
        }
```

evm6x_board_type

Retrieves Board Type and Version Information

Syntax

#include <evm6xdll.h>
BOOL **evm6x_board_type(**
HANDLE *h_device*,
PEVM6X_BOARD_TYPE *p_board_type*,
PULONG *p_rev_id***);**

Description

The evm6x_board_type() function retrieves the board type and revision ID information that is stored in the PCI configuration space of the board. The return value indicates the success of the function call.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The *p_board_type* and the *p_rev_id* parameters are pointers to the locations in which to place the requested information. After a successful evm6x_board_type() function call to a McEVM board is made, the location pointed to by the *p_board_type* parameter contains the enumerated value **TYPE_MCEVM**. The location pointed to by the *p_rev_id* parameter contains the board's revision ID as retrieved from the board's PCI configuration space. The board revision ID is used to indicate McEVM board hardware revisions.

Return Value

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

Example

In the following example, the evm6x_board_type() function retrieves information about the board type of the open board.

```
#include <windows.h>
#include <stdio.h>
#include <evm6xdll.h>
. . .
    HANDLE                  h_board;
    EVM6XDLL_BOARD_TYPE     t_board_type;
    ULONG                   ul_rev_id;
    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        exit(–1);
    }
    if ( !evm6x_board_type( h_board, &t_board_type,
        &ul_rev_id) )
    {
      printf( ”ERROR: evm6x_board_type() failed.\n” );
    }
    else
    {
        if ( t_board_type == TYPE_EVM )
        {
         printf( ”EVM Board, Revision %d.\n”, ul_rev_id );
        }
        else
        {
            printf( “Unknown board type.\n” );
        }
    }
```

evm6x_clear_message_event

Clears the Message Event

Syntax

#include <evm6xdll.h>
BOOL **evm6x_clear_message_event(**HANDLE *h_device***);**

Description

The evm6x_clear_message_event() function sets the message event to the nonsignaled state. Call this function to clear out any previous events before setting up to receive new events.

- ❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_clear_message_event() function clears the message event for a McEVM board to the nonsignaled state.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    if ( !evm6x_clear_message_event( h_board ) )
    {
        /* evm6x_clear_message_event failed */
        evm6x_close( h_board );
        exit(–1);
    }
. . .
```

evm6x_close *Closes a Driver Connection to a Board*

Syntax

#include <evm6xdll.h>
BOOL **evm6x_close(**HANDLE *h_device***);**

Description

The evm6x_close() function closes a previously opened driver connection to a board. The returned value is TRUE for a successful operation.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_close() function closes a previously opened driver connection to a McEVM board.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;
    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        exit(–1);
    }
. . .
    evm6x_close( h_board );
```

evm6x_coff_display

Displays COFF Information

Syntax

#include <evm6xdll.h>
BOOL **evm6x_coff_display(**
char **filename*,
BOOL *clear_bss_flag*,
BOOL *dump_flag***);**

Description

The evm6x_coff_display() function outputs COFF file information to stdout. 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, causes the bss section to be set to 0. This is not the default behavior of the DSP debugger.
- ❑ The *dump_flag* parameter, if TRUE, causes all of the data being written to DSP memory to be displayed to stdout. 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 evm6x_coff_display() function displays the section information of a COFF file to stdout.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    if ( !evm6x_coff_display( ”example.out”, FALSE, FALSE ) )
    {
        /* COFF display failed */
    }
```

evm6x_coff_load *Loads a COFF Image to a Board Using the HPI*

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_coff_load(
          HANDLE     h_device,
          LPVOID     lp_hpi,
          char       *filename,
          BOOL       verbose_flag,
          BOOL       clear_bss_flag,
          BOOL       dump_flag);
```

Description

The evm6x_coff_load() function reads a COFF image and writes the data to DSP memory using the HPI. This function allows you to load data or executable images from a COFF file to DSP memory.

- ❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- ❑ The *lp_hpi* parameter is either NULL or the handle returned from a successful evm6x_open_hpi() call. If this parameter is NULL, the function calls evm6x_open_hpi() to get its own handle to the HPI then closes it before returning. If the parameter is not NULL, the function uses the provided handle to the HPI and does not close it before returning. If the HPI is currently open, it cannot be opened again because the HPI supports only one user at a time.
- ❑ The *filename* parameter is the filename of the COFF file to be processed.
- ❑ The *verbose_flag* parameter, if TRUE, causes COFF file information to be sent to stdout during COFF file processing.
- ❑ The *clear_bss_flag* parameter, if TRUE, causes the bss section to be set to 0. This is not the default behavior of the DSP debugger.
- ❑ The *dump_flag* parameter, if TRUE, causes all of the data being written to DSP memory to be displayed to stdout. 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 evm6x_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.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

/*----------------------------------*
    reset DSP into HPI boot mode
    configure emif registers
    load COFF executable
    unreset DSP
 *----------------------------------*/
    evm6x_reset_dsp( h_board, HPI_BOOT );
    evm6x_init_emif( h_board, NULL );
    if ( !evm6x_coff_load( h_board, NULL, ”example.out”,
                           TRUE, FALSE, FALSE ) )
    {
        /* COFF load failed */
    }

    evm6x_unreset_dsp( h_board );
```

evm6x_cpld_read_all

Reads the Contents of the CPLD Registers

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_cpld_read_all(
     HANDLE    h_device,
     PULONG    p_reg_array);
```

Description

The evm6x_cpld_read_all()function reads all the CPLD registers and stores the values into the ULONG array provided by the user. Note that the low byte of each ULONG is the only portion that is significant since the CPLD registers are only 8 bits wide. This function is for testing and debugging purposes. General access to the CPLD registers is not required for user programs.

- ❑ The h_device parameter is the handle returned from a successful evm6x_open() call.
- ❑ The p_reg_array parameter is a pointer to the array of ULONGs that will be filled with the CPLD register contents. This array must be at least 9 elements in size to hold all the CPLD registers of the EVM board.

Return Value

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

Example

In the following example, the evm6x_cpld_read_all function is used to read the current state of the CPLD registers.

Opcode

```
#include <windows.h>
#include <evm6xdll.h>
. . .
   HANDLE    h_board;
   ULONG     reg_array[9];

   h_board = evm6x_open( 0, FALSE );
   if ( h_board == INVALID_HANDLE_VALUE )
   {
      /* unable to open board */
      exit(–1);
   }

   /* read the CPLD registers */
   if ( !evm6x_cpld_read_all( h_board, reg_array ) )
   {
      /* evm6x_cpld_read_all function failed */
   }
```

evm6x_dll_revision

Reads EVM DLL Revision Information

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_dll_revision(
     PULONG    p_RevMajor,
     PULONG    p_Rev_Minor,
     PULONG    p_BuildNum)
```

Description

The evm6x_dll_revisiont() function reads the revision information from the EVM DLL.

❑ The pRevMajor parameter is a pointer to where the major revision information of the DLL will be written.

❑ The pRevMinor parameter is a pointer to where the minor revision information of the DLL will be written.

❑ The pBuildNum parameter is a pointer to where the build number of the DLL will be written.

Return Value

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

evm6x_generate _nmi_int

Generates an NMI to a DSP

Syntax

#include <evm6xdll.h>
BOOL **evm6x_generate_nmi_int(**HANDLE *h_device***);**

Description

The evm6x_generate_nmi_int() function causes an NMI interrupt to be generated to the DSP. This interrupt can be disabled by the DSP.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_generate_nmi_int() function generates an NMI to the DSP on a McEVM board.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* send a NMI to the DSP */
    if ( !evm6x_generate_nmi_int( h_board ) )
    {
        /* evm6x_generate_nmi_int function failed */
    }
```

evm6x_hpi_close

Closes the HPI for a Board

Syntax

#include <evm6xdll.h>
BOOL **evm6x_hpi_close(**LPVOID *h_hpi_map***);**

Description

The evm6x_hpi_close() function closes an open HPI session that was started with a successful evm6x_hpi_open() call.

- ❑ The *h_hpi_map* parameter is the handle returned from a successful evm6x_hpi_open() call.

Return Value

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

Example

In the following example, the evm6x_hpi_close() function closes the HPI after reading from DSP memory.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    ULONG     ul_ret_len;
    ULONG     ul_buffer[2];

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(-1);
    }

    h_hpi = evm6x_hpi_open( h_board );
    if ( h_hpi == NULL )
    {
        /* evm6x_hpi_open() failed */
        evm6x_close( h_board );
        exit(-1);
    }

    /* read DSP memory (2 words, 32bits each) */
    ul_ret_len = 8;
    evm6x_hpi_read( h_hpi, ul_buffer, &ul_ret_len,
                    0x1f0 );

    if ( !evm6x_hpi_close( h_hpi ) )
    {
        /* evm6x_hpi_close failed */
    }
```

evm6x_hpi_fill *Fills DSP Memory Using the HPI*

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_hpi_fill(
        LPVOID      h_hpi_map,
        ULONG       fill_value,
        PULONG      p_length,
        ULONG       dest_addr);
```

Description

The evm6x_hpi_fill() function fills target DSP memory space with a fixed data value. The HPI is used to access DSP memory from the host.

- ❑ The *h_hpi_map* parameter is the handle returned from a successful evm6x_hpi_open() call.
- ❑ The *fill_value* parameter is the 32-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 because all transfers are 32-bit words.
- ❑ The *dest_addr* parameter is the fill starting address in the DSP's memory space. The address must be 32-bit word aligned. This address is from the DSP's point of view.

Return Value

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

Example

In the following example, the evm6x_hpi_fill() function fills 31 words (32 bits each) of DSP memory starting with address 0x1f84 with the data pattern 0x1234cdef.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    ULONG     ul_ret_len;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }
```

```
h_hpi = evm6x_hpi_open( h_board );
if ( h_hpi == NULL )
{
    /* evm6x_hpi_open() failed */
    evm6x_close( h_board );
    exit(–1);
}

/* fill 31 words of DSP memory (32bits each) */
ul_ret_len = 0x7c;
if ( !evm6x_hpi_fill( h_hpi, 0x1234cdef, &ul_ret_len,
       0x1f84 ) || (ul_ret_len != 0x7c) )
{
    /* evm6x_hpi_fill() failed */
}

evm6x_hpi_close( h_hpi );
```

evm6x_hpi_ generate_int

Generates an Interrupt to a DSP Using the HPI

Syntax

#include <evm6xdll.h>
BOOL **evm6x_hpi_generate_int(**LPVOID *h_hpi_map***);**

Description

The evm6x_hpi_generate_int() function causes an HPI interrupt (DSPINT) on the target DSP.

❑ The *h_hpi_map* parameter is the handle returned from a successful evm6x_hpi_open() call.

Return Value

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

Example

In the following example, the evm6x_hpi_generate_int() function generates an interrupt to a DSP using the HPI.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    h_hpi = evm6x_hpi_open( h_board );
    if ( h_hpi == NULL )
    {
        /* evm6x_hpi_open() failed */
        evm6x_close( h_board );
        exit(–1);
    }

    /* generate an HPI interrupt */
    if ( !evm6x_hpi_generate_int( h_hpi ) )
    {
        /* evm6x_hpi_generate_int() failed */
    }
. . .
```

evm6x_hpi_open

Opens the HPI for a Board

Syntax

#include <evm6xdll.h>
LPVOID **evm6x_hpi_open(**HANDLE *h_device***);**

Description

The evm6x_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 *h_device* parameter is the handle returned from a successful evm6x_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 an HPI interrupt to the DSP to be delayed.

Return Value

The function returns one of the following values:

Handle	Handle to be used for HPI access
NULL	Attempt to open the HPI failed

Example

In the following example, the evm6x_hpi_open() function opens a handle to the HPI on a McEVM board. This handle is required for access to DSP memory through the HPI using the evm6x_hpi_read(), evm6x_hpi_write(), and evm6x_hpi_fill() functions.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    ULONG     ul_ret_len;
    ULONG     ul_buffer[2];

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(-1);
    }
```

```
h_hpi = evm6x_hpi_open( h_board );
if ( h_hpi == NULL )
{
    /* evm6x_hpi_open() failed */
    evm6x_close( h_board );
    exit(-1);
}

/* read DSP memory (2 words, 32bits each) */
ul_ret_len = 8;
if ( !evm6x_hpi_read( h_hpi, ul_buffer, &ul_ret_len,
     0x1f0 ) ||
            (ul_ret_len != 8) )
{
    /* evm6x_hpi_read() failed */
}
evm6x_hpi_close( h_hpi );
```

evm6x_hpi_read — *Reads DSP Memory Using the HPI*

Syntax

#include <evm6xdll.h>
BOOL **evm6x_hpi_read(**
LPVOID *h_hpi_map*,
PULONG *p_buffer*,
PULONG *p_length*,
ULONG *src_addr***);**

Description

The evm6x_hpi_read() function transfers data from the target DSP memory space to host memory. The HPI is used to access DSP memory from the host.

- The *h_hpi_map* parameter is the handle returned from a successful evm6x_hpi_open() call.
- The *p_buffer* parameter is the address of the buffer to 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 4 because all transfers are 32-bit words.
- The *src_addr* parameter is the transfer starting address in the DSP's memory space. The address must be 32-bit word aligned. This address is from the DSP's point of view.

Return Value

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

Example In the following example, the evm6x_hpi_read() function reads two words (32 bits each) from DSP memory at address 0x1f0.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    ULONG     ul_ret_len;
    ULONG     ul_buffer[2];

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    h_hpi = evm6x_hpi_open( h_board );
    if ( h_hpi == NULL )
    {
        /* evm6x_hpi_open() failed */
        evm6x_close( h_board );
        exit(–1);
    }

    /* read DSP memory (2 words, 32bits each) */
    ul_ret_len = 8;
    if ( !evm6x_hpi_read( h_hpi, ul_buffer, &ul_ret_len,
          0x1f0 ) ||
                (ul_ret_len != 8) )
    {
        /* evm6x_hpi_read() failed */
    }

    evm6x_hpi_close( h_hpi );
```

evm6x_hpi_read _single

Reads a Single Byte Using the HPI

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_hpi_read_single(
     LPVOID     h_hpi_map,
     LPVOID     p_data, int i_size,
     ULONG      src_addr)
```

Description

The evm6x_hpi_read_single function reads a single 8–bit, 16–bit or 32–bit value from the target DSP memory space. The HPI is used to access DSP memory from the host. Please note that this call reads a 32–bit aligned value from DSP memory then returns the appropriate portion of the 32–bit value based on the address and size of the request. Reads smaller than 32–bits are not supported. Thus, if a read of all the bytes of a 32–bit aligned access will produce undesirable results, then this call should not be used.

- ❑ The h_hpi_map parameter is the handle returned from a successful evm6x_hpi_open call.
- ❑ The p_data parameter is the address of the location that will be filled by the read operation. This address must be aligned for the size of the HPI access requested.
- ❑ The i_size parameter is the size of the HPI access in bytes. It can be 1 for an 8–bit access, 2 for a 16–bit access or 4 for a 32–bit access.
- ❑ The src_addr parameter is the address in the DSP's memory space to be read. This address must be aligned to the requested access size. This address is from the DSP's point of view.

Return Value

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

Example

In the following example, the evm6x_hpi_read_single function is used to read a byte (8–bits), a short (16–bits) and a long (32–bits) from DSP memory at addresses 0x800001c3, 0x800001ca and 0x800001cc.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    UCHAR     uc_temp;
    USHORT    us_temp;
    ULONG     ul_temp;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    h_hpi = evm6x_hpi_open( h_board );
    if ( h_hpi == NULL )
    {
        /* evm6x_hpi_open() failed */
        evm6x_close( h_board );
        exit(–1);
    }

    /* read a byte from DSP memory (8–bits) */
    if ( !evm6x_hpi_read_single( h_hpi, &uc_temp, 1,
0x800001c3 ) )
    {
        /* evm6x_hpi_read_single() failed */
    }

    /* read a short from DSP memory (16–bits) */
    if ( !evm6x_hpi_read_single( h_hpi, &us_temp, 2,
0x800001ca ) )
    {
        /* evm6x_hpi_read_single() failed */
    }

    /* read a long from DSP memory (32–bits) */
    if ( !evm6x_hpi_read_single( h_hpi, &ul_temp, 4,
0x800001cc ) )
    {
        /* evm6x_hpi_read_single() failed */
    }
evm6x_hpi_close(h_hpi);
```

evm6x_hpi_write — *Writes to DSP Memory Using the HPI*

Syntax

#include <evm6xdll.h>
BOOL **evm6x_hpi_write(**

LPVOID	*h_hpi_map*,
PULONG	*p_buffer*,
PULONG	*p_length*,
ULONG	*dest_addr***);**

Description

The evm6x_hpi_write() function transfers data from host memory to the target DSP memory space. The HPI is used to access DSP memory from the host.

- The *h_hpi_map* parameter is the handle returned from a successful evm6x_hpi_open() call.
- The *p_buffer* parameter is the address of the buffer to 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 4 because all transfers are 32-bit words.
- The *dest_addr* parameter is the transfer starting address in the DSP's memory space. The address must be 32-bit word aligned. This address is from the DSP's point of view.

Return Value

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

Example

In the following example, the evm6x_hpi_write() function writes two words (32 bits each) to DSP memory at address 0x1f0.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    LPVOID    h_hpi;
    ULONG     ul_ret_len;
    ULONG     ul_buffer[2];

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }
```

```
h_hpi = evm6x_hpi_open( h_board );
if ( h_hpi == NULL )
{
    /* evm6x_hpi_open() failed */
    evm6x_close( h_board );
    exit(–1);
}

/* write 2 words to DSP memory (32bits each) */
ul_ret_len = 8;
ul_buffer[0] = 0x12345678;
ul_buffer[1] = 0xfedcba98;
if ( !evm6x_hpi_write( h_hpi, ul_buffer, &ul_ret_len,
      0x1f0 ) ||
            (ul_ret_len != 8) )
{
    /* evm6x_hpi_write() failed */
}

evm6x_hpi_close( h_hpi );
```

evm6x_hpi_write_single

Writes a Single Byte, Short or Long, Using the HPI

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_hpi_write_single(
    LPVOID    h_hpi_map,
    ULONG     ul_data, int i_size,
    ULONG     dest_addr);
```

Description

The evm6x_hpi_write_single function writes a single 8–bit, 16–bit or 32–bit data value to the target DSP memory space. The HPI is used to access DSP memory from the host. Please note that internal program memory does not support byte accesses. Thus, any write smaller than 32–bits to internal program memory will not modify memory as expected.

- The h_hpi_map parameter is the handle returned from a successful evm6x_hpi_open call.
- The ul_data parameter contains the value to be written to DSP memory.
- The low 8–bits or 16–bits of the value will be used for those size accesses.
- The i_size parameter is the size of the HPI access in bytes. It can be 1 for an 8–bit access, 2 for a 16–bit access or 4 for a 32–bit access.
- The dest_addr parameter is the address in the DSP's memory space to be accessed. This address must be aligned to the requested access size. This address is from the DSP's point of view.

Return Value

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

Example

In the following example, the evm6x_hpi_write_single function is used to write a byte (8–bits), a short (16–bits) and a long (32–bits) to DSP memory at addresses 0x800001f1, 0x800001f6 and 0x800001e0.

Opcode

```
#include <windows.h>
#include <evm6xdll.h>
. . .
   HANDLE    h_board;
   LPVOID    h_hpi;
   ULONG     ul_temp;

   h_board = evm6x_open( 0, FALSE );
   if ( h_board == INVALID_HANDLE_VALUE )
   {
      /* unable to open board */
```

```
        exit(–1);
    }

    h_hpi = evm6x_hpi_open( h_board );
    if ( h_hpi == NULL )
    {
        /* evm6x_hpi_open() failed */
        evm6x_close( h_board );
        exit(–1);
    }

    /* write a byte (8–bits) to DSP memory */
    ul_temp = 0x3f;
    if ( !evm6x_hpi_write_single( h_hpi, ul_temp, 1, 0x800001f1 ) )
    {
        /* evm6x_hpi_write_single() failed */
    }

/* write a short (16–bits) to DSP memory */
    ul_temp = 0xbeef;
    if ( !evm6x_hpi_write_single( h_hpi, ul_temp, 2, 0x800001f6 ) )
    {
        /* evm6x_hpi_write_single() failed */
    }

/* write a long (32–bits) to DSP memory */
    ul_temp = 0x87654321;
    if ( !evm6x_hpi_write_single( h_hpi, ul_temp, 4, 0x800001e0 ) )
    {
        /* evm6x_hpi_write_single() failed */
    }

    evm6x_hpi_close( h_hpi );
```

evm6x_init_emif *Initializes the EMIF Registers*

Syntax

#include <evm6xdll.h>
BOOL **evm6x_init_emif(**
HANDLE *h_device*,
LPVOID *lp_hpi***);**

Description

The evm6x_init_emif() function sets the EMIF registers using HPI accesses to the DSP memory space. This operation is used in conjunction with HPI boot mode. Call the evm6x_reset_dsp() function to reset the DSP and put it into HPI boot mode (see page 2-40). Then, before loading code to memory, call the evm6x_init_emif() function to initialize the EMIF registers so that external memory on the board is accessible via the HPI.

- The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- The *lp_hpi* parameter is the handle returned from a successful evm6x_hpi_open() call or NULL if the HPI is not already open. If this parameter is NULL, the HPI is opened for this operation then closed at the completion of the function. If this parameter is not NULL, the value is used as the handle for HPI accesses and is not closed at the completion of the function.

Return Value

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

Example

In the following example, the evm6x_init_emif() function properly configures the EMIF registers of the DSP for the McEVM board. This step must be done after resetting the DSP and before the DSP tries to access memory on the McEVM board.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }
```

```
/*----------------------------------*
    reset DSP into HPI boot mode
    configure emif registers
    load COFF executable
    unreset DSP
 *----------------------------------*/
    evm6x_reset_dsp( h_board, HPI_BOOT );
    if ( !evm6x_init_emif( h_board, NULL ) )
    {
        /* evm6x_init_emit call failed */
    }

    evm6x_coff_load( h_board, NULL, ”example.out”,
                     FALSE, FALSE, FALSE );
    evm6x_unreset_dsp( h_board );
```

evm6x_mailbox_read

Reads From an EVM Mailbox

Syntax

#include <evm6xdll.h>
BOOL **evm6x_mailbox_read(**
HANDLE *h_device*,
ULONG *regNum*,
PULONG *p_data***);**

Description

The evm6x_mailbox_read() function reads information from the EVM’s mailbox register.

- The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- The regNum parameter represents the mailbox number to be read.
- The pData parameter is a pointer to the location in which the data will be stored.

Return Value

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

evm6x_mailbox_write

Writes To an EVM Mailbox

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_mailbox_write(
          HANDLE    h_device,
          ULONG     regNum,
          ULONG     data);
```

Description

The evm6x_mailbox_write() function writes information to the EVM's mailbox register.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The regNum parameter represents the target mailbox number.

❑ The data parameter is the data to be written.

Return Value

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

evm6x_nvram_read

Reads a Byte of NVRAM

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_nvram_read(
          HANDLE    h_device,
          USHORT    offset,
          PUCHAR    p_data);
```

Description

The evm6x_nvram_read() function reads a byte from a target board's NVRAM.

> **Avoiding Simultaneous NVRAM Accesses**
>
> Make sure that NVRAM read and/or write operations do not happen at the same time from both the host and the DSP. Simultaneous accesses to NVRAM will result in invalid operations and will possibly corrupt data stored in NVRAM.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The *offset* parameter is the NVRAM byte offset to be read.

❑ The *p_data* parameter is a pointer to the location in which to place the byte read from NVRAM.

Return Value

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

Example

In the following example, the evm6x_nvram_read() function reads the byte value stored in NVRAM at offset 0x83.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    UCHAR     c_data;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    if ( !evm6x_nvram_read( h_board, 0x83, &c_data ) )
    {
        /* evm6x_nvram_read failed */
    }
    else
    {
        /* byte read from offset 0x83 is in c_data */
    }
```

evm6x_nvram_write

Writes a Byte to NVRAM

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_nvram_write(
        HANDLE     h_device,
        USHORT     offset,
        UCHAR      data);
```

Description

The evm6x_nvram_write() function writes a byte to a target board's NVRAM. This write operation is not allowed to the lower 0x80 bytes of NVRAM. This space is used for PCI configuration settings and should not be modified.

CAUTION

Avoiding Simultaneous NVRAM Accesses

Make sure that NVRAM read and/or write operations do not happen at the same time from both the host and the DSP. Simultaneous accesses to NVRAM will result in invalid operations and will possibly corrupt data stored in NVRAM.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The *offset* parameter is the NVRAM byte offset to be written.

❑ The *data* parameter is the byte data value to be written to NVRAM.

Return Value

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

Example

In the following example, the evm6x_nvram_write() function writes the byte value 0x3f to NVRAM at offset 0xc3.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;
    UCHAR     c_data;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    c_data = 0x3f;
    if ( !evm6x_nvram_write( h_board, 0xc3, c_data ) )
    {
        /* evm6x_nvram_write failed */
    }
```

evm6x_open — *Opens a Driver Connection to a Board*

Syntax

#include <evm6xdll.h>
HANDLE **evm6x_open(**
int *board_index*,
BOOL *exclusive_flag***);**

Description

The evm6x_open() function opens a driver connection to a specific EVM or McEVM board. 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 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 one of the following values:

HANDLE	Handle to be used for all further accesses to the target board
INVALID_HANDLE_VALUE	Operation failed

Example

In the following example, the evm6x_open() function performs a nonexclusive open of the first EVM board in a system.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    int      board_index;
    HANDLE   h_board;

    board_index = 0;  /* select the first (or only) EVM */
                     /* board                          */

    h_board = evm6x_open( board_index, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* board open failed */
        exit(–1);
    }
. . .
    evm6x_close( h_board );
```

evm6x_read

Reads Data from a Board

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_read(
        HANDLE      h_device,
        PULONG      p_buffer,
        PULONG      p_length);
```

Description

The evm6x_read() function transfers data from the DSP to the host using the PCI bus mastering capability of the board. This transfer can take an indeterminate amount of time, depending on the DSP making data available for the transfer. To prevent the host from waiting too long for a transfer, a time-out feature is available. Use the evm6x_set_timeout() function to set the time-out value (see page 2-46).

Also, a transfer can be terminated at any time using the evm6x_abort_read() function (see page 2-7). A transfer that has been timed out or terminated still returns a success indication, but the length value returned is not the same as the originally requested transfer length.

- The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- The *p_buffer* parameter is the address of the buffer to 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 4 because all transfers are 32-bit words.

Return Value

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

Example

In the following example, the evm6x_read() function transfers 292 bytes (73 words, 32 bits each) from the DSP into host memory using the PCI FIFO interface provided on the McEVM board. The transfer may succeed, but be incomplete if the transfer timed out or a transfer abort request was initiated from a separate thread. For additional information, see the evm6x_set_timeout() function description on page 2-46 and the evm6x_abort_read() function description on page 2-7.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;
    ULONG    ul_length;         /* requested transfer */
                                /* length in bytes    */
    ULONG    ul_ret_len;        /* returned transfer  */
                                /* length in bytes    */
    ULONG    ul_buffer[1024];   /* data buffer, must  */
                                /* be 32bit aligned   */

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    ul_length = 292;
    ul_ret_len = ul_length;
    if ( !evm6x_read( h_board, ul_buffer, &ul_ret_len ) )
    {
        /* evm6x_read failed. */
    }
    else
    {
        if ( ul_ret_len != ul_length )
        {
            /* evm6x_read incomplete */
            /* this can be the result of a time out or */
            /* an abort                              */
        }
    }
```

evm6x_reset_board

Resets a Board

Syntax

#include <evm6xdll.h>
BOOL **evm6x_reset_board(**HANDLE *h_device***);**

Description

The evm6x_reset_board() function causes a hardware reset pulse on the target board. The target board must be opened exclusively using the evm6x_open() function (see page 2-36) or this function will fail. The return value indicates the success of the function call.

❑ The *h_device* parameter is the handle returned from a successful exclusive evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_reset_board() function resets the second McEVM board in the system. This board was previously opened exclusively.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;

    /* open the second EVM board exclusively */
    h_board = evm6x_open( 1, TRUE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board exclusively */
        exit(–1);
    }

    if ( !evm6x_reset_board( h_board ) )
    {
        /* reset call failed */
    }
    else
    {
        /* reset call succeeded */
    }
```

evm6x_reset_dsp *Resets a DSP*

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_reset_dsp(
          HANDLE                    h_device,
          EVM6XDLL_BOOT_MODE   mode);
```

Description

The evm6x_reset_dsp() function sets the boot mode and causes a reset pulse to only the DSP. If the boot mode is set to HPI boot, the DSP is left in a halted state. To allow the DSP to begin executing, use the evm6x_unreset_dsp() function (see page 2-47). The return value indicates the success of the function call.

- The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- The enumerated *mode* parameter is the boot mode in which to reset the DSP.
  - HPI_BOOT resets the DSP into HPI boot mode with the MAP 1 memory map.
  - HPI_BOOT_MAP0 resets into HPI boot mode with MAP 0.
  - NO_BOOT selects no boot mode with MAP 1.
  - NO_BOOT_MAP0 selects no boot mode with MAP 0.
  - The various ROMX_BOOT boot modes require that an executable boot image is present on a daughterboard.

The usual procedure for loading and starting a DSP is:

1) Reset the DSP in HPI boot mode using the evm6x_reset_dsp() function.

2) Configure the DSP memory interface with HPI write calls to access the EMIF registers. This can be accomplished with the evm6x_init_emif() function (see page 2-31).

3) Load the DSP program with HPI write calls using the evm6x_coff_load() function (see page 2-13).

4) Unreset the DSP from its halted state using the evm6x_unreset_dsp() function (see page 2-47).

Return Value

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

Example

In the following example, the the evm6x_reset_dsp() function places the DSP into HPI boot mode. While in this state, the DSP accepts HPI accesses. The evm6x_unreset_dsp() function releases the DSP from this state and begins execution of DSP code. See the evm6x_unreset_dsp() function example on page 2-47 for more information.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE                  h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }
    if ( !evm6x_reset_dsp(h_board, HPI_BOOT) )
    {
        /* DSP reset call failed */
    }
    else
    {
        /* DSP reset call succeeded */
    }
```

evm6x_retrieve_message

Retrieves a Message from a DSP

Syntax

#include <evm6xdll.h>
BOOL **evm6x_retrieve_message(**
HANDLE *h_device*,
PEVM6X_MESSAGE *p_message***);**

Description

The evm6x_retrieve_message() function retrieves a mailbox message from the target DSP. This function returns FALSE if the mailbox to the host from the DSP is not full.

This is a completely independent mailbox from that used by the evm6x_send_message() function. Also, the receipt of a message from the DSP can be detected by a Win32 event. The interrupt caused by an incoming message signals an event. The name of the event signaled is the string defined in EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME with the board index value appended as a decimal number.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The *p_message* parameter is a pointer to the location to place the message from the DSP.

Return Value

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

Example

In the following example, the evm6x_retrieve_message() function checks for and retrieves a 32-bit word sent by the DSP after waiting for the Win32 event signaling a message arrival. If the evm6x_retrieve_message() function fails, then a message has not been sent by the DSP. However, this should not happen in this example because the routine waits for the event signaling a new message before retrieving the message.

```
#include <windows.h>
#include <stdio.h>
#include <evm6xdll.h>
. . .
    HANDLE              h_board;
    HANDLE              h_event;
    EVM6XDLL_MESSAGE    t_message;
    int                 n_board_index;
    char                s_buffer[80];

    n_board_index = 1;
    h_board = evm6x_open( n_board_index, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* create event name and open handle to the event */
    sprintf( s_buffer, ”%s%d”,
             EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME,
             n_board_index );
    h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );

    /* wait for event signaling a message from the DSP */
    if ( WaitForSingleObject( h_event, INFINITE )
        != WAIT_OBJECT_0 )
    {
        /* wait for event failed */
    }

    /* retrieve message sent by DSP */
    if ( !evm6x_retrieve_message( h_board, &t_message ) )
    {
        /* evm6x_retrieve_message() failed */
        /* this means that no message is available */
    }
    else
    {
     /* the 32bit value sent by the DSP is in t_message */
    }
. . .
```

evm6x_send_message

Sends a Message to a DSP

Syntax

#include <evm6xdll.h>
BOOL **evm6x_send_message(**
HANDLE *h_device*,
PEVM6X_MESSAGE *p_message***);**

Description

The evm6x_send_message() function sends a mailbox message to the target DSP. This function returns FALSE if the mailbox from the host to the DSP is not empty.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

❑ The *p_message* parameter is a pointer to the message to be sent to the DSP.

Return Value

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

Example

In the following example, the evm6x_send_message() function sends a 32-bit value, 0xfeed002f, to a DSP using the mailbox capability of the McEVM board.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE              h_board;
    EVM6XDLL_MESSAGE    t_message;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* send message to DSP */
    t_message = 0xfeed002f;
    if ( !evm6x_send_message( h_board, &t_message ) )
    {
        /* evm6x_send_message() failed */
    }
. . .
```

evm6x_set_ board_config

Sets the User Board Options

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_set_board_config(
         HANDLE                     h_device,
         EVM6XDLL_CLOCK_MODE        e_clock_mode,
         EVM6XDLL_ENDIAN_MODE       e_endian_mode,
         ULONG                      user_bits );
```

Description

The evm6x_set_board_config() function sets the software configuration for the board. This includes the clock mode, the endian mode, and the user bits. This operation is not required if the hardware DIP switch settings are to be used. If software settings are to be used, this operation must be done before the DSP is reset using the evm6x_reset_dsp() function (see page 2-40).

- ❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- ❑ The *e_clock_mode* enumerated parameter is one of these values:
  - ■ DSP_CLOCK_SBSRAM configures the DSP to run at the maximum speed for SBSRAM accesses
  - ■ DSP_CLOCK_NORMAL configures the DSP to run at the normal clock speed.
  - ■ DSP_CLOCK_AX1 configures the DSP to run at the oscillator A multiply-by-1 clock speed.
  - ■ DSP_CLOCK_BX1 configures the DSP to run at the oscillator B multiply-by-1 clock speed.
- ❑ The *e_endian_mode* enumerated parameter is one of these values:
  - ■ LITTLE_ENDIAN_MODE configures the DSP to run in little-endian mode.
  - ■ BIG_ENDIAN_MODE configures the DSP to run in big-endian mode.
- ❑ The *user_bits* parameter is the value to be used for the three user bits. If the value is greater than 7, the value is not used. In this case, the hardware DIP switch setting for the user bits is used.

Return Value

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

Example

In the following example, the evm6x_set_board_config() function sets the McEVM board configuration, overriding the hardware DIP switch settings. This must be done before resetting the DSP. In this example, the clock mode is set to DSP_CLOCK_SBSRAM mode, which configures the DSP clock to the speed of the SBSRAM. This allows the DSP to access SBSRAM at one clock per access. The endian mode is set to BIG_ENDIAN_MODE, which allows the DSP to execute software compiled and linked to run on a big-endian DSP. The user bits are set to 0xff. The user bits set on the hardware DIP switch are used because the user bits value is greater than 7.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

/*----------------------------------*
    set board configuration
    reset DSP into HPI boot mode
    configure emif registers
    load COFF executable
    unreset DSP
 *----------------------------------*/
    if ( !evm6x_set_board_config( h_board,
                                 DSP_CLOCK_SBSRAM,
                                 BIG_ENDIAN_MODE, 0xff ) )
    {
        /* evm6x_set_board_config call failed */
    }

    evm6x_reset_dsp( h_board, HPI_BOOT );
    evm6x_init_emif( h_board, NULL );
    evm6x_coff_load( h_board, NULL, ”example.out”,
                     FALSE, FALSE, FALSE );
    evm6x_unreset_dsp( h_board);
```

evm6x_set_timeout

Sets the Transfer Time-Out

Syntax

#include <evm6xdll.h>
BOOL **evm6x_set_timeout(**
    HANDLE *h_device*,
    ULONG *timeout***);**

Description

The evm6x_set_timeout() function sets the time-out value for the PCI bus mastering data transfers. If a transfer exceeds this time-out period, the transfer is terminated. The transfer return value indicates how much of the data was transferred.

❑ The *h_device* parameter is the handle returned from a successful evm6x_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 evm6x_set_timeout() function sets the bus master transfer time-out value to 5000 ms (or 5 s). This means that any transfer that takes longer than 5 s will terminate, returning the actual number of bytes transferred during the transfer. A transfer stalls when data to or from the DSP is not available.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

    /* set the transfer timeout to 5 seconds */
    if ( !evm6x_set_timeout( h_board, 5000 ) )
    {
        /* evm6x_set_timeout failed */
    }
```

evm6x_unreset_dsp

Unresets a DSP After Using HPI Boot Mode

Syntax

#include <evm6xdll.h>
BOOL **evm6x_unreset_dsp(**HANDLE *h_device***);**

Description

The evm6x_unreset_dsp() function releases the DSP from the halted state invoked by the evm6x_reset_dsp() function with the *mode* parameter set to HPI boot (see page 2-40). Use this function in conjunction with an evm6x_reset_dsp() call only. The return value indicates the success of the function call.

❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.

Return Value

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

Example

In the following example, the evm6x_unreset_dsp() function releases the DSP from the halted state that results from resetting the DSP into HPI boot mode.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE    h_board;

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }

/*----------------------------------*
    reset DSP into HPI boot mode
    configure emif registers
    load COFF executable
    unreset DSP
 *----------------------------------*/

    evm6x_reset_dsp( h_board, HPI_BOOT );
    evm6x_init_emif( h_board, NULL);
    evm6x_coff_load( h_board, NULL, ”example.out”,
                     FALSE, FALSE, FALSE );

    if ( !evm6x_unreset_dsp( h_board ) )
    {
        /* DSP unreset call failed */
    }
```

evm6x_user_ semaphore_get

Acquires User Semaphore

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_user_semaphore_get(
     HANDLE    h_device,
     PULONG    semState);
```

Description

The evm6x_user_semaphore_get function is used to acquire the user semaphore.

❑ The semState parameter is a pointer to the semaphore.

❑ The h_device parameter is the handle returned from a successful evm6x_open call.

Return Value

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

evm6x_user_ semphore release

Releases User Semaphore

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_user_semaphore_release(
     HANDLE    h_device,);
```

Description

The evm6x_user_semaphore_release function is used to release the user semaphore.

❑ The h_device parameter is the handle returned from a successful evm6x_open call.

Return Value

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

evm6x_user_ semphore wait

Waits Until the Semaphore is Available

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_user_semaphore_wait(
    HANDLE    h_device,);
```

Description

The evm6x_user_semaphore_wait function is used to wait until the semaphore is available.

- ❑ The h_device parameter is the handle returned from a successful evm6x_open call.

Return Value

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

evm6x_write

Writes Data to a Board

Syntax

```
#include <evm6xdll.h>
BOOL evm6x_write(
        HANDLE    h_device,
        PULONG    p_buffer,
        PULONG    p_length);
```

Description

The evm6x_write() function transfers data from host memory to the DSP using the PCI bus mastering capability of the board. This transfer can take an indeterminate amount of time, depending on the DSP accepting data for the transfer. To prevent the host from waiting too long for a transfer, a time-out feature is available. Use the evm6x_set_timeout() function to set the time-out value (see page 2-46).

Also, a transfer can be terminated at any time using the evm6x_abort_write() function (see page 2-8). A transfer that has been timed out or terminated still returns a success indication, but the length value returned is not the same as the originally requested transfer length.

- ❑ The *h_device* parameter is the handle returned from a successful evm6x_open() call.
- ❑ The *p_buffer* parameter is the address of the buffer to 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 4 because all transfers are 32-bit words.

Return Value

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

Example

In the following example, the evm6x_write() function sends 292 bytes (73 words, 32 bits each) from host memory to the DSP using the PCI FIFO interface provided on the McEVM board. The transfer may succeed, but be incomplete if the transfer timed out or a transfer abort request was initiated from a separate thread. For additional information, see the evm6x_set_timeout() function description on page 2-46 and the evm6x_abort_write() function description on page 2-8.

```
#include <windows.h>
#include <evm6xdll.h>
. . .
    HANDLE   h_board;
    ULONG    ul_length;        /* requested transfer */
                               /* length in bytes    */
    ULONG    ul_ret_len;       /* returned transfer  */
                               /* length in bytes    */
    ULONG    ul_buffer[1024];  /* data buffer, must  */
                               /* be 32bit aligned   */

    h_board = evm6x_open( 0, FALSE );
    if ( h_board == INVALID_HANDLE_VALUE )
    {
        /* unable to open board */
        exit(–1);
    }
. . .
    ul_length = 292;
    ul_ret_len = ul_length;
    if ( !evm6x_write( h_board, ul_buffer, &ul_ret_len ) )
    {
        /* evm6x_write failed. */
    }
    else
    {
        if ( ul_ret_len != ul_length )
        {
            /* evm6x_write incomplete */
            /* this can be the result of a time out or */
            /* an abort                               */
        }
    }
```

2.4 McEVM Host Support Software Example

The following is a simple program that loads and runs a COFF file in the McEVM. The program illustrates the use of a number of the most basic McEVM Win32 DLL calls.

Example 2–2. McEVM Win32 DLL Sample Code

```
/*-------------------------------------------------------------------------*/
/* FILENAME: HostApp.c –– Host Support Software Example                    */
/*-------------------------------------------------------------------------*/
#include <windows.h>
#include ”evm6xdll.h”
void WriteWord2Mem( LPVOID, ULONG, ULONG );
/*-------------------------------------------------------------------------*/
/* main()                                                                  */
/*-------------------------------------------------------------------------*/
void main(int argc1, char *argv1[])
{
  HANDLE hBd    = NULL; /* Board device handle                             */
  short  iBd    = 0;    /* Board index                                     */
  BOOL   bExcl  = 1;    /* Exclusive open = TRUE                           */
  short  iMp    = 0;    /* Map selector = MAP0                             */
  EVM6XDLL_BOOT_MODE mode = HPI_BOOT_MAP0;
                        /* DSP boot mode                                   */
  LPVOID hHpi   = NULL; /* HPI interface handle                            */
  char coffNam[]= ”blink.out”;
                        /* COFF file name                                  */
  BOOL bVerbose = 0;    /* COFF load verbose mode = FALSE                  */
  BOOL bClr     = 0;    /* Clear bss mode = FALSE                          */
  BOOL bDump    = 0;    /* Dump mode = FALSE                               */
  /*-----------------------------------------------------------------------*/
  /* Open a driver connection to a specific [Mc]EVM6x board.               */
  /*-----------------------------------------------------------------------*/
  hBd = evm6x_open( iBd, bExcl );
  if ( hBd == INVALID_HANDLE_VALUE ) exit(1);
  /*-----------------------------------------------------------------------*/
  /* Cause a hardware reset on the target board.                           */
  /*-----------------------------------------------------------------------*/
  if ( !evm6x_reset_board(hBd) ) exit(2);
  /*-----------------------------------------------------------------------*/
  /* Set the boot mode and cause a DSP reset.                              */
  /*-----------------------------------------------------------------------*/
  mode = iMp ? HPI_BOOT : HPI_BOOT_MAP0;
  if ( !evm6x_reset_dsp(hBd,mode) ) exit(3);
  /*-----------------------------------------------------------------------*/
  /* Establish a connection to the HPI of a target board.                  */
  /*-----------------------------------------------------------------------*/
  hHpi = evm6x_hpi_open(hBd);
  if ( hHpi == NULL ) exit(4);
  /*-----------------------------------------------------------------------*/
  /* Initialize EMIF registers                                             */
  /*-----------------------------------------------------------------------*/

  if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

  /*-----------------------------------------------------------------------*/
  /* set Aux DMA priority higher than CPU                                  */
  /*-----------------------------------------------------------------------*/
  /*  Due to the default priority of the auxiliary DMA channel used for    */
  /*  HPI accesses, the CPU can prevent HPI accesses from completing for   */
  /*  an indeterminate amount of time.  This can occur when the CPU is     */
  /*  very active on the external memory interface, such as while executing*/
  /*  code from external memory.  This condition manifests itself as a     */
  /*  hung PCI bus.  To prevent this condition, the value 0x10 can be      */
  /*  written to the DMA Auxiliary Control Register of the 6201 (at        */
  /*  address 0x01840070).  This elevates the priority of the auxiliary    */
  /*  DMA channel above all other DMA channels and above the CPU.          */
  /*-----------------------------------------------------------------------*/
  WriteWord2Mem( hHpi, 0x01840070 /*Addr*/, 0x00000010 /*Data*/ );
  /*-----------------------------------------------------------------------*/
  /* Read a COFF file and write the data to DSP memory.                    */
  /*-----------------------------------------------------------------------*/
  if (!evm6x_coff_load(hBd,hHpi,coffNam,bVerbose,bClr,bDump)) exit(8);
  /*-----------------------------------------------------------------------*/
  /* Close the HPI session started with evm6x_hpi_open()                   */
  /*-----------------------------------------------------------------------*/
  if (!evm6x_hpi_close(hHpi)) exit(9);
  /*-----------------------------------------------------------------------*/
  /* Release the DSP from the halted state                                 */
  /*-----------------------------------------------------------------------*/
  if (!evm6x_unreset_dsp(hBd)) exit(10);
  /*-----------------------------------------------------------------------*/
  /* Close a previously opened driver connection to a board.               */
  /*-----------------------------------------------------------------------*/
  if (!evm6x_close(hBd)) exit(11);
  exit(0);
} /* end of main()                                                         */
/*-------------------------------------------------------------------------*/
/*  Write one word (32 bits) to DSP memory                                 */
/*-------------------------------------------------------------------------*/
void WriteWord2Mem( LPVOID hHpi, ULONG ulDataAddr, ULONG ulDataWord )
{
  ULONG       ulLength;
  ULONG       ulReturnedLength;
  ulLength = 4;
  ulReturnedLength = ulLength;
  if ( !evm6x_hpi_write( hHpi, &ulDataWord,
                               &ulReturnedLength, ulDataAddr) ) exit(6);
  if ( ulLength != ulReturnedLength ) exit(7);
}
```

Chapter 3

TMS320C62x McEVM DSP Support Software

This chapter describes the McEVM DSP support software by providing application programming interfaces (APIs) and example code for the multichannel buffered serial port (McBSP) driver, Multi-Vendor Integration Protocol (MVIP) library, voice-band audio processor (VBAP) library, T1/E1 framer library**,** and board support library. All of these modules use the TMS320C6000 peripheral support library to access and control internal peripheral registers. See the *TMS320C6x Peripheral Support Library Programmer's Reference* for a description of this library.

Topic **Page**

3.1 DSP Support Software Components

The DSP support software consists of these components:

❑ McBSP driver
❑ MVIP library
❑ Board support library
❑ FMIC library
❑ T1/E1 framer library
❑ VBAP library
❑ PCI/AMCC library
❑ C I/O device library

The example code provided operates the MT90810 flexible MVIP integrated circuit (FMIC) in serial data mode, and all telephony data is communicated to and from the ’C62x via the McBSP0. Configuration and control of the MVIP FMIC are achieved via the parallel data interface. These two modules are totally independent of one another in that the MVIP library makes no calls to the McBSP driver and vice-versa. It is your responsibility to configure both peripherals for use and to control them independently. The board support library, on the other hand, is used by both the McBSP driver and the MVIP library.

The McBSP provides two types of routines for sending and receiving data. The *synchronous routines* mcbsp_sync_send() and mcbsp_sync_receive() transfer data by polling the data transmit ready (DXR) and data receive ready (DRR) bits, respectively, to determine when the next word can be written or read. These routines are referred to as blocking because the CPU is blocked waiting for the transfer to complete before returning control to the caller. The *asynchronous routines* mcbsp_async_send() and mcbsp_async_receive() use the direct memory access (DMA) engine of the ’C62x to transfer the data in the background. User-supplied callback functions are invoked upon completion of the data transfers to signal the caller. These callback functions are invoked from an interrupt service routine and may set a global flag, for instance, which would indicate data is ready to be processed.

The MVIP library is a collection of routines that configure and control the operation of the MT90810 MVIP FMIC. The API functions correspond closely to the functional organization of the chip. Numerous macros are defined for this library in the file MT90810.h. These macros can be used as arguments to the API functions.

The board support library provides routines for configuring and controlling the McEVM and returning status information to the caller. The module also includes utility functions such asLED control and delay routines.

The FMIC library is a collection of routines that configure and control the operation of the MT90810 FMIC device. The API functions provide low level access, which allows user code to configure the device into any desired configuration.

The API functions also provide higher level access for use in standard device operations.

The T1/E1 Framer library is a collection of routines that configure and control the operation of the Siemens FALC55 (PEB2255) device. The API functions provide low level access, which allows user code to configure the device into any desired configuration. The API functions also provide higher level access for use in standard device operations

The VBAP library is a collection of routines that control the selection between the μ-law (North American & Japanese) and the A-law (European) companding devices.

The PCI/AMCC library is a collection of routines that provide host communications via the AMCC S5933 device. This driver provides data streams using PCI Bus Mastering and message passing using mailboxes.

The C I/O device library is a collection of routines that provide the required interface between the TI C Compiler's C I/O support and the PCI/AMCC driver. Using this C I/O device library functions and the add_device() function from the TI C Compiler, DSP code can send and receive data streams with the host using standard C I/O functions such as fprintf() and fgets().

3.2 Using the DSP Support Software Components

The DSP support software, which consists of the McBSP driver, MVIP library, VBAP library, T1/E1 framer library, and board support library, is installed from the accompanying CD-ROM to the \evm6x\dsp\lib\drivers directory. These components are in object format and are supplied in the archived object library file drv6x.lib. Another file, drv6xe.lib, is the big-endian version of the archived object library. The source for the DSP support software is contained in the source library file drv6x.src. To extract the source files, assuming that you have installed the ’C62x code generation tools, enter:

```
ar6x –x drv6x.src
```

This command extracts the two makefiles in the drv6x.src file. The files makefile and makefile.big are the little- and big-endian makefile versions, respectively. The first two lines of these files must be modified to point to your c6xtools directory and to your ’C6000 peripheral support library files. See the *TMS320C6x Peripheral Support Library Programmer’s Reference* for more information about building this library in the desired mode.

To build the object files from the extracted source (*.c) files, enter one of these commands:

```
nmake                                  For little-endian object modules
nmake –fmakefile.big                   For big-endian object modules
```

To rebuild the object library, enter one of these commands:

```
nmake drv6x.lib                        For little-endian object library
nmake –fmakefile.big drv6xe.lib        For big-endian object library
```

It is possible to build the file drv6xe.lib with little-endian object files and vice-versa, so use caution when building these libraries. Any attempt to use a library of one endian version with code of another will produce a linker error.

Example code that uses the McBSP driver, MVIP library, VBAP library, T1/E1 framer library, and board support library exists in the .\evm6x\dsp\examples directory. Again, the files makefile and makefile.big refer to the little- and big-endian makefiles, respectively. The first two lines of these makefiles must also be modified to point to your c6xtools and ’C6000 peripheral support library files. These makefiles also provide an example of how to include the drv6x.lib file on the linker command line for user-developed code. See the *TMS320C6x Assembly Language Tools User’s Guide* for more information about using object libraries.

The provided example code configures and uses the MVIP FMIC, VBAP, and T1/E1 framer devices and the McBSP in numerous configurations, such as loopback, block capture and playback, continuous capture and playback using ping-pong buffering, and continuous tone generation. You can run the example code via the ’C62x McEVM debugger, or you can load the code and allow it to run using the McEVM COFF loader utility (evm6xldr). The print statements that are visible in the debugger command window are not visible on the DOS screen when you use the COFF loader.

3.3 McBSP Driver API

This section discusses the McBSP driver API. Included in this discussion are the macros, data types, and defined functions that comprise the McBSP driver for the ’C62x McEVM board.

3.3.1 McBSP Driver Macros

Table 3–1 lists the macros defined, their values, and a description of each. The API functions that use each macro are also listed. These macros are defined within the public header file mcbspdrv.h. The majority of the macros used by the McBSP driver are defined within the ’C62x device library header file mcbsp.h. These macros are used as bit and bit field values within registers and correspondingly within elements of the driver data types. See the *TMS320C6x Peripheral Support Library Programmer’s Reference* and the following section for further information.

Table 3–1. McBSP Driver API State Macros

Macro	Value	Description
CLOSED	1	Flags McBSP as closed
OPEN	2	Flags McBSP as open and waiting for control
BUSY	3	Flags McBSP as busy

Note: These macros are used by all McBSP driver functions.

3.3.2 McBSP Driver Data Types

This section lists the public data types defined by the McBSP driver that are required for accessing McBSP driver functions.

Example 3–1. McBSP Driver API Data Types

(a) McBSP device handle

```
typedef   Mcbsp_handle * Mcbsp_dev;
```

An initialized Mcbsp_handle is required for all subsequent McBSP driver calls. The data structure Mcbsp_dev is used as a private (global static) structure that records state information for each port.

(b) McBSP callback function

```
typedef void Callback(Mcbsp_dev  dev,  int  status);
```

Callback functions are used by mcbsp_async_send() and mcbsp_async_receive() to indicate completion of the requested action.

(c) McBSP configuration structure

```
typedef struct Mcbsp_config_struct
{
  int                  loopback;
  Mcbsp_tx_config      tx;
  Mcbsp_rx_config      rx;
  Mcbsp_srg_config     srg;
}
Mcbsp_config;
```

The Mcbsp_config structure is used when calling the mcbsp_config() function. The element loopback is TRUE or FALSE and is used to control the data loopback mode. The transmitter, receiver, and sample rate generator configuration values are contained in the three structure elements listed in Example 3–1(d), Example 3–1(e), and Example 3–1(f), respectively.

Example 3–1. McBSP Driver API Data Types (Continued)

(d) McBSP transmitter configuration structure

```
typedef struct Mcbsp_tx_config_struct

{
  unsigned char   update;              /* Update Tx Parameters? T/F            */
  unsigned char   interrupt_mode;      /* SPCR(2): XRDY,Blk,Frame,SyncErr      */
  unsigned char   clock_polarity;      /* PCR(1): Rise or Fall of CLKX         */
  unsigned char   frame_sync_polarity; /* PCR(1): Active High or Low           */
  unsigned char   clock_mode;          /* PCR(1): External or Internal         */
  unsigned char   frame_sync_mode;     /* PCR(1): External or Internal         */
  unsigned char   phase_mode;          /* XCR(1): Single or Dual               */
  unsigned char   frame_length1;       /* XCR(7): 1 to 128 wpf for phase 1     */
  unsigned char   frame_length2;       /* XCR(7):    ”      ”      phase 2     */
  unsigned char   word_length1;        /* XCR(3): bits per phase 1 word        */
  unsigned char   word_length2;        /* XCR(3): bits per phase 2 word        */
  unsigned char   companding;          /* XCR(2): ALAW ULAW or MSB or LSB      */
  unsigned char   frame_ignore;        /* XCR(1): T/F                          */
  unsigned char   data_delay;          /* XCR(3): 0–2 Tx data delay            */
  unsigned char   dummy[2];            /* pad bytes to 32 bit align            */
}

Mcbsp_tx_config;
```

The Mcbsp_tx_config structure is a substructure of Mcbsp_config and is used to assign register values for the transmitter. The corresponding bits and bit field values are shown to the right of each structure element. See the *TMS320C6201/C6701 Peripherals Reference Guide* for a detailed discussion of each value.

Example 3–1. McBSP Driver API Data Types (Continued)

(e) McBSP receiver configuration structure

```
typedef struct Mcbsp_rx_config_struct

{
  unsigned char   update;              /* Update Rx Parameters? T/F            */
  unsigned char   interrupt_mode;      /* SPCR(2): RRDY,Blk,Frame,Syncerr      */
  unsigned char   justification;       /* SPCR(2): RJZF, RJSE or LJZF          */
  unsigned char   clock_polarity;      /* PCR(1): Rise or Fall of CLKX         */
  unsigned char   frame_sync_polarity; /* PCR(1): Active High or Low           */
  unsigned char   clock_mode;          /* PCR(1): External or Internal         */
  unsigned char   frame_sync_mode;     /* PCR(1): External or Internal         */
  unsigned char   phase_mode;          /* XCR(1): Single or Dual               */
  unsigned char   frame_length1;       /* XCR(7):1 to 128 wpf for phase 1      */
  unsigned char   frame_length2;       /* XCR(7):    ″      ″      phase 2     */
  unsigned char   word_length1;        /* XCR(3): bits per phase 1 word        */
  unsigned char   word_length2;        /* XCR(3): bits per phase 2 word        */
  unsigned char   companding;          /* XCR(2): ALAW ULAW or MSB or LSB      */
  unsigned char   frame_ignore;        /* XCR(1): T/F                          */
  unsigned char   data_delay;          /* XCR(3): 0–2 Rx data delay            */
  unsigned char   dummy;               /* pad bytes                            */
}

Mcbsp_rx_config;
```

The Mcbsp_rx_config structure is a substructure of Mcbsp_config and is used to assign register values for the receiver. The corresponding bits and bit field values are shown to the right of each structure element. See the *TMS320C6201/C6701 Peripherals Reference Guide* for a detailed discussion of each value.

(f) McBSP sample rate generator configuration structure

```
typedef struct Mcbsp_srg_config_struct

{
  unsigned char   update;            /* Update SRGR Parameters? T/F      */
  unsigned char   clock_sync;        /* SRGR(1):GSYNC_OFF or GSYNC_ON    */
  unsigned char   clks_polarity;     /* SRGR(1):rising or falling edge   */
  unsigned char   clks_mode;         /* SRGR(1):external or internal     */
  unsigned char   frame_sync_mode;   /* SRGR(1):FSX due to DXR–XSR, FSG  */
  unsigned short  frame_period;      /* SRGR(12): Frame period 1–4096    */
  unsigned char   frame_width;       /* SRGR(8): 1 to 256 CLKG periods   */
  unsigned char   clock_divider;     /* SRGR(8): SRGR clock dvdr: 1–256  */
}

Mcbsp_srg_config;
```

The Mcbsp_srg_config structure is a substructure of Mcbsp_config and is used to assign register values for the receiver. The corresponding bits and bit field values are shown to the right of each structure element. See the *TMS320C6201/C6701 Peripherals Reference Guide* for a detailed discussion of each value.

3.3.3 McBSP Driver Functions

The following alphabetical listing includes all of the McBSP driver API functions. Use this listing as a table of contents to the McBSP driver API functions.

mcbsp_async_receive

Receives a Buffer of Data on the Selected McBSP Asynchronously

Syntax

```
#include <mcbspdrv.h>
int  mcbsp_async_receive(
              Mcbsp_dev         dev,
              unsigned char     *p_buffer,
              unsigned int      num_bytes,
              unsigned int      frame_sync_enable,
              Mcbsp_dev         frame_sync_dev,
              Callback          *p_ract);
```

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_async_receive() function receives a buffer of data from the indicated McBSP in an asynchronous (also known as nonblocking) manner. This routine transfers data in the background using an available direct memory access (DMA) engine. An interrupt service routine is set up by this routine; once the indicated number of bytes have been transferred, the BLOCK COND bit in the DMA secondary control register triggers the enabled DMA interrupt. This interrupt service routine can call a user-supplied callback function, which could, for example, be used to set a global transfer flag indicating that reception is finished. You do not have access to the interrupt service routine, only the callback function to which you provide a pointer.

❑ Parameter *dev* refers to the initialized device handle returned by the mcbsp_open() call.

❑ The *p_buffer* parameter is a pointer to the buffer used to hold received data.

❑ The *num_bytes* parameter refers to the number of bytes to receive. Typically, the McBSP is configured for 32-bit transfers. In this case, *num_bytes* is four times the number of 32-bit elements to receive. You must ensure that the buffer pointed to by *p_buffer* is at least as large as *num_bytes*.

❑ The *frame_sync_enable* parameter is used to enable the internal frame sync generator. Valid values are TRUE or FALSE.

❑ The *frame_sync_dev* parameter indicates which port's frame sync generator to enable. If *frame_sync_enable* is FALSE, set *frame_sync_dev* to NULL. In most cases, *frame_sync_dev* is the same as *dev*.

❑ The *p_ract* parameter is a pointer to the user-supplied callback function. See section 3.3.2, *McBSP Driver Data Types*, for the typedef Callback, which defines the structure of the callback function.

Return Value

The function returns one of the following values:

OK	Transfer setup succeeded
ERROR	Transfer setup failed

Example

The following code provides an example invocation assuming that the driver has been initialized and the device has been opened and configured for an external frame sync and 32-bit transfers.

```
#define NUM_WORDS     128

Callback  callback_function(Mcbsp_dev dev, int status)
{
 printf("Data transfer for mcbsp_async_receive() completed
         with status = %d\n",status);
 return;
}

int          status;
unsigned int buffer[NUM_WORDS];

status= mcbsp_async_receive(dev0,
                            buffer,
                            NUM_WORDS * sizeof(int),
                            FALSE,
                            NULL,
                            callback_function
                           );
if (status == ERROR)
{
 printf("Error setting up data transfer with
         mcbsp_sync_receive()\n");
 return(ERROR);
}
```

mcbsp_async_send

Sends a Buffer of Data on the Selected McBSP Asynchronously

Syntax

```
#include <mcbspdrv.h>
int mcbsp_async_send(
            Mcbsp_dev         dev,
            unsigned char     *p_buffer,
            unsigned int      num_bytes,
            unsigned int      frame_sync_enable,
            Mcbsp_dev         frame_sync_dev,
            Callback          *p_wact);
```

Defined in mcbspdrv.c as a callable C routine

Description The mcbsp_async_send() function sends a buffer of data from the indicated McBSP in an asynchronous (also known as nonblocking) manner. This routine transfers data in the background using an available DMA engine. An interrupt service routine is set up by this routine; once the indicated number of bytes have been transferred, the BLOCK COND bit in the DMA secondary control register triggers the enabled DMA interrupt. This interrupt service routine can call a user-supplied callback function, which could, for example, be used to set a global transfer flag indicating that transmission is finished. You do not have access to the interrupt service routine, only the callback function to which you provide a pointer.

❏ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

❏ The *p_buffer* parameter is a pointer to the buffer used to hold data to be transmitted.

❏ The *num_bytes* parameter refers to the number of bytes to send. Typically, the McBSP is configured for 32-bit transfers. In this case, *num_bytes* is four times the number of 32-bit elements to receive.

❏ The *frame_sync_enable* parameter is used to enable the internal frame sync generator. Valid values are TRUE or FALSE.

❏ The *frame_sync_dev* parameter indicates which port's frame sync generator to enable. If *frame_sync_enable* is FALSE, set *frame_sync_dev* to NULL. In most cases, *frame_sync_dev* is the same as *dev.*

❏ The *p_wact* parameter is a pointer to the user-supplied callback function. See section 3.3.2, *McBSP Driver Data Types*, for the typedef Callback, which defines the structure of the callback function.

Return Value

The function returns one of the following values:

OK	Transfer setup succeeded
ERROR	Transfer setup failed

Example

The following code provides an example invocation assuming that the driver has been initialized and the device has been opened and configured for an internal frame sync and 32-bit transfers.

```
#define NUM_WORDS     128

Callback  callback_function(Mcbsp_dev dev, int status)
{
 printf(“Data transfer for mcbsp_async_send() completed
        with status = %d\n”,status);
 return;
}

int          status;
unsigned int buffer[NUM_WORDS];

status= mcbsp_async_send(dev0,
                         buffer,
                         NUM_WORDS * sizeof(int),
                         TRUE,
                         dev0,
                         callback_function
                        );
if (status == ERROR)
{
 printf(“Error setting up data transfer with \
        mcbsp_sync_send()\n”);
 return(ERROR);
}
```

mcbsp_close

Closes the Selected McBSP/Release the Device Handle

Syntax

#include <mcbspdrv.h>
void **mcbsp_close(**Mcbsp_dev *dev***);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_close() function closes the selected McBSP and releases its associated device handle structure.

❑ The *dev* parameter is the device handle that was initialized by the mcbsp_open() call.

Return Value

None

Example

The code in this example closes the McBSP associated with *dev***,** allowing another routine or thread to control its operation.

```
mcbsp_close(dev);
```

mcbsp_config *Configures the Selected McBSP*

Syntax

```
#include <mcbspdrv.h>
int  mcbsp_config(
          Mcbsp_dev        dev,
          Mcbsp_config     *p_mcbsp_config);
```

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_config() function configures the selected McBSP. Configuration values are passed to this routine via the Mcbsp_config structure (see section 3.3.2, *McBSP Driver Data Types*, for the definition of this structure). The macro defines for the configuration structure elements are defined in the file mcbsp.h, which is part of the ’C6000 device library.

❑ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

❑ The *p_mcbsp_config* parameter is a pointer to a user-initialized McBSP configuration structure.

Return Value

The function returns one of the following values:

OK	Configuration values are within range of their respective min/max values.
ERROR	Configuration values were greater than or less than their respective min/max values.

Example

The code in this example configures the transmitter, receiver, and sample rate generator for the McBSP associated with the initialized device handle *dev* for use in internal loopback mode. Defines shown in this example are part of the ’C6000 peripheral control library.

```
config.loopback              = TRUE;
config.tx.update             = TRUE;
config.tx.clock_polarity     = CLKX_POL_RISING;
config.tx.frame_sync_polarity= FSYNC_POL_HIGH;
config.tx.clock_mode         = CLK_MODE_INT;
config.tx.frame_sync_mode    = FSYNC_MODE_INT;
config.tx.phase_mode         = SINGLE_PHASE;
config.tx.frame_length1      = 0;
config.tx.word_length1       = WORD_LENGTH_32;
config.tx.frame_ignore       = NO_FRAME_IGNORE;
config.tx.data_delay         = DATA_DELAY1;
config.rx.update             = TRUE;
config.rx.clock_polarity     = CLKR_POL_FALLING;
config.rx.frame_sync_polarity= FSYNC_POL_HIGH;
config.rx.clock_mode         = CLK_MODE_EXT;
config.rx.frame_sync_mode    = FSYNC_MODE_EXT;
config.rx.phase_mode         = SINGLE_PHASE;
config.rx.frame_length1      = 0;
config.rx.word_length1       = WORD_LENGTH_32;
config.rx.frame_ignore       = NO_FRAME_IGNORE;
config.rx.data_delay         = DATA_DELAY1;
config.srg.update            = TRUE;
config.srg.clks_mode         = CLK_MODE_INT;
config.srg.frame_sync_mode   = FSX_DXR_TO_XSR;
config.srg.frame_width       = 1;
config.srg.clock_divider     = 0xff;
if (mcbsp_config(dev0,&config))
{
  printf(”Error returned from mcbsp_config() for dev0”);
  mcbsp_close(dev0);
  return(ERROR);
}
```

mcbsp_cont_async_send

Continuously Sends a Buffer of Data on the Selected McBSP

Syntax

#include <mcbspdrv.h>
int **mcbsp_cont_async_send(**

Mcbsp_dev	*dev*,
unsigned char	**p_ping_buff*,
unsigned char	**p_pong_buff*,
unsigned int	*num_bytes*,
unsigned int	*frame_sync_enable*,
int	*McBSP_dev frame_sync_dev*,
Callback	**p_wact***);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_cont_async_send() function repeatedly sends either a single buffer or a ping-pong type buffer in an asynchronous (also known as nonblocking) manner. This routine transfers data in the background using an available DMA engine. An interrupt service routine is set up by this routine; once the indicated number of bytes have been transferred, the BLOCK COND bit in the DMA secondary control register triggers the enabled DMA interrupt. This interrupt service routine can call a user-supplied callback function, which could, for example, be used to set a global transfer flag indicating another transmission is finished. You do not have access to the interrupt service routine, only the callback function to which you provide a pointer.

❑ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

❑ The *p_ping_buffer* parameter is a pointer to the buffer used to hold data to be transmitted.

- The *p_pong_buffer* parameter is a pointer to the secondary transmit buffer. If this parameter is NULL, the data in *p_ping_buffer* is continuously sent. If this parameter is valid, data transmission ping-pongs between these two buffers.
- The *num_bytes* parameter refers to the number of bytes to send (same for each buffer). Typically, the McBSP is configured for 32-bit transfers. In this case, *num_bytes* is four times the number of 32-bit elements to receive.
- The *frame_sync_enable* parameter is used to enable the internal frame sync generator. Valid values are TRUE or FALSE.
- The *frame_sync_dev* parameter indicates which port's frame sync generator to enable. If *frame_sync_enable* is FALSE, set *frame_sync_dev* to NULL. In most cases, *frame_sync_dev* is the same as *dev*.
- The *p_wact* parameter is a pointer to the user-supplied callback function. See section 3.3.2, *McBSP Driver Data Types*, for the typedef Callback, which defines the structure of the callback function.

Return Value

The function returns one of the following values:

OK	Transfer setup succeeded
ERROR	Transfer setup failed

Example

The following code provides an example invocation assuming that the driver has been initialized and the device has been opened and configured for an internal frame sync and 32-bit transfers. This code continuously sends 16 buffers of data, eight from the *ping_buff[ ]* and eight from the *pong_buff[ ]*. To turn off continuous operation, the callback function must set dev–>tx_dma.continuous to FALSE one block before transmission should stop.

```
#define NUM_WORDS     128
#define NUM_BUFFS     16

int     num_buffs_sent=    0;

Callback  callback_function(
               Mcbsp_dev dev,
               int       status)
{
 printf(“Data transfer for mcbsp_async_send() completed
         with\ status = %d\n”,status);
 num_buffs_sent++;

 if (num_buffs_sent == NUM_BUFFS)
   mcbsp_stop(dev);

 return;
}

int          status;
unsigned int ping_buff[NUM_WORDS];
unsigned int pong_buff[NUM_WORDS];
```

```
for (i=0;i<NUM_WORDS;i++)  /* create a ramp */
  ping_buff[i] = i;

for (i=0;i<NUM_WORDS;i++)  /* create a ramp */
  pong_buff[i] = i;

status= mcbsp_cont_async_send( dev0,
                               ping_buff,
                               pong_buff,
                               NUM_WORDS * sizeof(int),
                               TRUE,
                               dev0,
                               callback_function
                              );

if (status == ERROR)
{
 printf("Error setting up data transfer with \
         mcbsp_sync_send()\n");
 return(ERROR);
}
```

mcbsp_drv_init

Initializes the McBSP Driver

Syntax

#include <mcbspdrv.h>
int **mcbsp_drv_init(**void**);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_drv_init() function initializes the McBSP driver for use and must be called before any other driver calls. This function allocates memory for the port 0 and port 1 device handles and initializes structure elements to their default values.

Return Value

The function returns one of the following values:

OK	Memory allocation succeeded
ERROR	Memory allocation failed

Subsequent calls to this function simply return OK.

Example

The code in this example initializes the McBSP driver for use and is called before any other McBSP routines.

```
int status;
status = mcbsp_drv_init();
if (status == ERROR)
{
  printf("Error initializing the McBSP driver\n");
  return(ERROR);
}
```

mcbsp_open — *Opens the Selected McBSP/Obtain a Device Handle*

Syntax

#include <mcbspdrv.h>
Mcbsp_dev **mcbsp_open(**int *port***);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_open() function opens the selected McBSP for use. All subsequent driver calls require an initialized *Mcbsp_dev* argument.

❑ The *port* parameter indicates which serial port to open, with valid values of 0 or 1.

Return Value

The function returns one of the following values:

Mcbsp_dev	A valid device handle value is returned if the indicated *port* is not already in use.
NULL	The indicated *port* is already in use.

Example

The code in this example obtains a device handle for the McBSP port 0. This device handle is required in subsequent McBSP calls, such as mcbsp_config() and mcbsp_async_receive().

```
Mcbsp_dev dev0;
dev0 = mcbsp_open(0);
if (dev0 == NULL)
{
  printf(“Unable to obtain a device handle for McBSP 0.”);
  return(ERROR);
}
```

mcbsp_reset

Resets the Selected McBSP

Syntax

#include <mcbspdrv.h>
void **mcbsp_reset(**Mcbsp_dev *dev***);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_reset() function simply resets the selected McBSP associated with *dev* to its default state.

- ❑ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

Return Value

The function returns one of the following values:

OK	The *dev* parameter was supplied as a valid device handle.
ERROR	The *dev* parameter was not supplied as a valid device handle.

Example

The code in this example resets the associated McBSP to its default state.

```
if ( mcbsp_reset( dev ) )
{
  printf(“Invalid device handle passed to mcbsp_reset\n”);
  return(ERROR);
}
```

mcbsp_stop

Stops Operation of the Selected McBSP

Syntax

#include <mcbspdrv.h>
void **mcbsp_stop(**Mcbsp_dev *dev***);**

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_stop() function completely disables the selected McBSP. Both the transmit and receive sections are disabled, as well as the sample rate and frame sync generators.

- ❑ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

Return Value

None

Example

The code in this example disables the selected McBSP.

```
mcbsp_stop(dev);
```

mcbsp_sync_receive

Receives a Buffer of Data on the Selected McBSP Synchronously

Syntax

```
#include <mcbspdrv.h>
int mcbsp_sync_receive(
            Mcbsp_dev         dev,
            unsigned char     *p_buffer,
            unsigned int      num_bytes,
            unsigned int      frame_sync_enable,
            Mcbsp_dev         frame_sync_dev,
            int               pack_data);
```

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_sync_receive() function receives a buffer of data from the indicated McBSP in a synchronous (also known as blocking) manner. During data reception, the CPU is busy polling the data receive ready (DRR) flag in the serial port control register to determine when the next data word is available. Once the indicated number of bytes have been transferred to the buffer, this routine returns to the caller.

❑ The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

❑ The *p_buffer* parameter is a pointer to the buffer used to hold received data.

❑ The *num_bytes* parameter refers to the number of bytes to receive. Typically, the McBSP is configured for 32-bit transfers. In this case, *num_bytes* is four times the number of 32-bit elements to receive. You must ensure that the buffer pointed to by *p_buffer* is at least as large as *num_bytes*.

❑ The *frame_sync_enable* parameter is used to enable the internal frame sync generator. Valid values are TRUE or FALSE.

❑ The *frame_sync_dev* parameter indicates which port's frame sync generator to enable. If *frame_sync_enable* is FALSE, set *frame_sync_dev* to NULL. In most cases, *frame_sync_dev* is the same as *dev*.

❑ Parameter *pack_data* can be used to pack received elements that are less than 32 bits into *p_buffer*. Valid values for pack_data are TRUE or FALSE. The *pack_data* feature can only be used when the serial port is configured for single-phase transfers.

Return Value

The function returns one of the following values:

OK	Transfer succeeded
ERROR	Transfer failed

Example

The following code provides an example invocation assuming that the driver has been initialized and the device has been opened and configured for an external frame sync and 32-bit transfers.

```
#define NUM_WORDS     128

int          status;
unsigned int buffer[NUM_WORDS];

status= mcbsp_sync_receive(   dev0,
                              buffer,
                              NUM_WORDS * sizeof(int),
                              FALSE,
                              NULL,
                              FALSE);

if (status == ERROR)
{
 printf(“Error receiving data with mcbsp_sync_receive()\n”);
 return(ERROR);
}
```

mcbsp_sync_send

Sends a Buffer of Data on the Selected McBSP Synchronously

Syntax

```
#include <mcbspdrv.h>
int  mcbsp_sync_send(
          Mcbsp_dev         dev,
          unsigned char     *p_buffer,
          unsigned int      num_bytes,
          unsigned int      frame_sync_enable,
          Mcbsp_dev         frame_sync_dev,
          int               packed_data);
```

Defined in

mcbspdrv.c as a callable C routine

Description

The mcbsp_sync_send() function sends a buffer of data across the indicated McBSP in a synchronous (also known as blocking) manner. During data transmission, the CPU is busy polling the data transmit ready (DXR) flag in the serial port control register to determine when the next element can be written. Once the indicated number of bytes have been transferred from the buffer, this routine returns to the caller.

- The *dev* parameter refers to the initialized device handle returned by the mcbsp_open() call.

- The *p_buffer* parameter is a pointer to the buffer initialized with the data to send.

- The *num_bytes* parameter refers to the number of bytes to send. Typically, the McBSP is configured for 32-bit transfers. In this case, *num_bytes* is four times the number of 32-bit elements to send.
- The *frame_sync_enable* parameter is used to enable the internal frame sync generator. Valid values are TRUE or FALSE.
- The *frame_sync_dev* parameter indicates which port's frame sync generator to enable. If *frame_sync_enable* is FALSE, set *frame_sync_dev* to NULL. In most cases, *frame_sync_dev* is the same as *dev*.
- The *packed_data* parameter, if TRUE, indicates that the data in *p_buffer* is packed. For element transfers of less than 32 bits, this routine sends only the significant bits for each transfer, sign extended and justified as indicated in the mcbsp_config() call (see page 3-15).

Return Value

The function returns one of the following values:

OK	Transfer succeeded
ERROR	Transfer failed

Example

The following code provides an example invocation assuming that the driver has been initialized and the device has been opened and configured for an internal frame sync (FSG) and 32-bit transfers.

```
#define NUM_WORDS     128

int          status;
unsigned int buffer[NUM_WORDS];

for (i=0;i<NUM_WORDS;i++)     /* generate ramp */
  buffer[i]= i;

status= mcbsp_sync_send( dev0,
                         buffer,
                         NUM_WORDS * sizeof(int),
                         TRUE,
                         dev0,
                         FALSE);

if (status == ERROR)
{
 printf("Error sending data with mcbsp_sync_send()\n");
 return(ERROR);
}
```

3.4 FMIC Support Library API

This section discusses the Flexible MVIP Interface Circuit (FMIC) library. Included in this discussion are the macros, data types, and defined functions that comprise the FMIC driver for the ’C62x McEVM board.

3.4.1 FMIC Driver Macros

Table 3–2 lists the macros defined, their values, and a description of each. The SPI functions that use each macro are also listed. These macros are defined within the public header file fmic.h.

Table 3–2. FMIC Driver API State Macros

Macro	Value	Description
FMIC_MVIP_SLAVE	0	FMIC as timing slave
FMIC_T1E1_MVIP_MASTER	1	FMIC as timing master
FMIC_8KMASTER_MVIP_SLAVE	2	FMIC as timing slave
FMIC_8KMASTER_MVIP_MASTER	3	FMIC as timing master
FMIC_8KMASTER_MVIP_MASTER	4	FMIC as timing master
Note: These macros are used by Fmic_init and Fmic_timing.		
FMIC_DSo_IN	0	DSo selected as input for specified channel
FMIC_DSi_IN	1	DSi selected as input for specified channel
Note: These macros are used by Fmic_init and Fmic_direction.		
FMIC_MASTER_REG_ID		FMIC master control/status register
FMIC_CNTRL_REG_ID		FMIC control register
FMIC_DATA_MEMORY_ID		FMIC data memory
FMIC_CONN_MEM_HIGH_ID		FMIC connection memory – high byte
Note: These macros are used by Fmic_configure, fmic_read, and fmic_write.		

3.4.2 FMIC Driver Data Types

This section lists the public data types defined by the FMIC driver that are required for accessing FMIC driver functions.

Example 3–2. FMIC Driver API Data Types

(a) FMIC device handle

```
typedef   FMIC_DEV  *FMIC_DEV_T;
```

An initialized FMIC_DEV handle is required for all subsequent FMIC driver calls. The data structure FMIC_DEV_T is used as a private (global static) structure that records state information for each device.

(b) FMIC configuration structure

```
typedef struct

{
  fmic_state  state;
  u32 base_addr;

 }

FMIC_DEV;
```

The FMIC private structure records state information for each device.

3.4.3 FMIC Library API Functions

The following alphabetical listing includes all of the FMIC library API functions. Use this listing as a table of contents to the FMIC library API functions.

fmic_clear_ connections

Clears All Connections, Disable All MVIP Channel Outputs

Syntax

#include <fmic.h>
int **fmic_clear_connections(**FMIC_DEV_T fmic_dev**);**

Defined in

fmic.c as a callable C routine

Description

This function breaks all input output connections by clearing the connection memory for all MVIP and local channels for the specified FMIC device.

❑ The fmic_dev parameter is a pointer to a structure containing addressing and state information for the FMIC device.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example clears all connections on FMIC device dev0.

```
int status;

status = fmic_clear_connections(dev0);
if (status == ERROR)
   return(ERROR);
```

fmic_close

De-initializes Software Driver

Syntax

#include <fmic.h>
int **fmic_close(**FMIC_DEV_T fmic_dev**);**

Defined in

fmic.c as a callable C routine

Description

This function removes the handle to the specified FMIC device and resets device information to a default condition.

❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example The following example releases the FMIC device dev0.

```
int status;
status = fmic_close(dev0);
if (status == ERROR)
   return(ERROR);
```

fmic_configure *Configures an FMIC Device*

Syntax

```
#include ”fmic.h”
int fmic_configure(
        FMIC_DEV_T      fmic_dev,
        unsigned int    fmic_configuration[][3]);
```

Defined in fmic.c as a callable C routine

Description This function configures the specified FMIC device given an input array of register, value and mask sets. The function will write each configuration set in the array to the FMIC device until the mask element becomes zero. This will terminate the configuration.

- Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.
- Fmic_configuration is an array of three element sets consisting of the following information:
  - Fmic_configuration[ ][0] = register_id
  - Fmic_configuration[ ][1] = value
  - Fmic_configuration[ ][2] = mask
  - While mask is non-zero, write the configuration to the FMIC.
  - See the fmic_write description for possible values of register_id, value and mask.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following is a generalized example of the use of the fmic_configure function. It illustrates making multiple configuration changes with one call. It also illustrates the use of direct and indirect addressing for register_id (see fmic_write() for further description). The first configuration command sets the RESET bit in the master Control/Status register without affecting any other bits. The second configuration command inverts the 4.096MHz CLK4 clock output pin by setting bit 3 of the Local Clock Control register (indirect offset 1 in the FMIC control register). The third and fourth configuration commands make a connection from output 56 to input 34 while enabling output 56.

```
int status;
unsigned int  fmic_configuration[5][3] = {
   {FMIC_MASTER_REG_ID, 0x1, 0x1 },
   {FMIC_CNTRL_REG_ID | 1, 0x4,  0x4 },
   {FMIC_CONN_MEM_LOW_ID | 56, 34, 0xff},
   {FMIC_CONN_MEM_HIGH_ID | 56, 2, 0xff},
   {0,0,0}
};

status = fmic_configure(dev0, fmic_configuration);
if (status == ERROR)
   return(ERROR);
```

fmic_connect

Connects an Input Channel to an Output Channel

Syntax

#include ”fmic.h”
int **fmic_connect(**
FMIC_DEV_T *fmic_dev,*
unsigned int *out_chan,*
unsigned int *in_chan***)**;

Defined in

fmic.c as a callable C routine

Description

This function makes a connection between one of 384 input channels and one of 384 output channels on the specified FMIC. For a bidirectional connection, two separate calls should be made reversing the input/output channel combination on the second call.

- ❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.

❑ Out_chan is a zero relative input channel number from 0 to 383.

❑ In_chan is a zero relative output channel number from 0 to 383.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example makes a bidirectional connection between channels 56 and 34.

```
int status=0;
status = fmic_connect(dev0, 56, 34);
status |= fmic_connect(dev0, 34, 56);
if (status == ERROR)
   return(ERROR);
```

fmic_direction *Selects MVIP Inbound Source For the Specified Channel*

Syntax

#include ”fmic.h”
int **fmic_direction(**
FMIC_DEV_T *fmic_dev,*
unsigned int *channel,*
fmic_mvip_direction *direction***);**

Defined in

fmic.c as a callable C routine

Description

This function sets the direction of an associated DSi-DSo channel pair. If the DSi or DSo channel is programmed as an input, the corresponding DSo or DSi channel will automatically be configured as an output.

❑ The fmic_dev parameter is a pointer to a structure containing addressing and state information for the FMIC device.

❑ The channel parameter is the channel number from 0 to 255.

❑ The direction parameter selects the inbound source, (DSi, DSo) for the specified channel. It can be one of the following values.

- FMIC_DSo_IN selects DSo as input for the specified channel.
- FMIC_DSi_IN selects DSi as input for the specified channel.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example selects DSo as the input for channel 56. Consequently, DSi will automatically be set as the output for channel 56.

```
int status;
status = fmic_direction(dev0, 56, FMIC_DSo_IN);
if (status == ERROR)
   return(ERROR);
```

fmic_init

Establishes Normal Operating Register and Memory Values

Syntax

#include ”fmic.h”

int **fmic_init(**
FMIC_DEV_T *fmic_dev,*
fmic_timing_mode *timing_mode***)**;

Defined in

fmic.c as a callable C routine

Description

This function initializes the specified FMIC device by: returning all register values to reset condition, clearing connection and data ram, establishing normal operating register values and establishing requested timing mode.

❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.

❑ Timing_mode selects which clock source is used to generate MVIP clock signals. It can be one of the following values.

  ■ FMIC_MVIP_SLAVE selects the FMIC as timing slave to the MVIP bus.

  ■ FMIC_T1E1_MVIP_MASTER sele

  ■ FMIC_8KMASTER_MVIP_SLAVE selects the FMIC as timing slave to the MVIP bus, *****

  ■ FMIC_8KSLAVE_MVIP_MASTER selects the FMIC as timing master.

  ■ FMIC_T1E1_8K_MVIP_MASTER selects the FMIC as timing master,

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example initializes FMIC device dev0 and sets the timing mode to FMIC_MVIP_SLAVE.

```
int status;
status = fmic_init(dev0, FMIC_MVIP_SLAVE);
if (status == ERROR)
   return(ERROR);
```

fmic_open

Initializes the Software Driver for an FMIC Device

Syntax

#include ”fmic.h”
FMIC_DEV_T **fmic_open(**unsigned int *fmic_base_add**r)***;*

Defined in

fmic.c as a callable C routine

Description

This function creates a handle to the FMIC device located at the specified address.

❑ Fmic_base_addr is the base address for the FMIC device to be opened.

Return Value

Returns nonzero value as handle.

Example

The following example opens the FMIC device on a McEVM board.

```
FMIC_DEV_T dev0;
dev0 = fmic_open(FMIC_MOTHER_BASE);
if (dev0 == NULL)
   return(ERROR);
```

fmic_output_control

Enables MVIP Output Driver for the Specified Channel

Syntax

#include ”fmic.h”
int **fmic_output_control**
(FMIC_DEV_T *fmic_dev,*
unsigned int *channel,*
boolean *enable***)**;

Defined in

fmic.c as a callable C routine

Description

This function enables and disables output for the specified MVIP channels on the specified FMIC device.

❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.

❑ Channel specified the output channel from 0 to 255 on the MVIP bus to disable or enable

❑ Enable specifies whether the indicated channel is to be enabled. The possible values are:

  ■ TRUE

  ■ FALSE

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example enables the output for channel 56 on the FMIC device dev0.

```
int status;
status = fmic_output_control(dev0, 56, TRUE);
if (status == ERROR)
   return(ERROR);
```

fmic_read

Reads the Value at the FMIC Memory/Register Location

Syntax

#include ”fmic.h”
unsigned int **fmic_read(**

```
        FMIC_DEV_T      fmic_dev,
        unsigned int    register_id);
```

Defined in

fmic.c as a callable C routine

Description

This function returns the value contained in the indicated FMIC memory/register location on the specified FMIC device.

- ❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.
- ❑ Register_id is an identifier for either a direct or indirect register/memory location on the FMIC device. For an indirect location, the following indirect IDs should be OR'ed with the desired location in the register/memory space indicated to obtain the value to pass in as register_id. For example, to obtain the low byte of the 4th element in the connection memory, register_id = FMIC_CONN_MEM_LOW_ID | 3. A direct register does not need an offset and can be used as is. The possible register/memory IDs are defined as follows.
  - ■ FMIC_MASTER_REG_ID selects the master control/status register (direct).
  - ■ FMIC_CNTRL_REG_ID selects the FMIC control registers (indirect).
  - ■ FMIC_DATA_MEMORY_ID selects the data memory (indirect).
  - ■ FMIC_CONN_MEM_LOW_ID selects the low byte of the connection memory (indirect).
  - ■ FMIC_CONN_MEM_HIGH_ID selects the high byte of the connection memory (indirect).

Return Value

Returns the value at the given register_id.

Example

The following example reads the value in the master control register. It also reads and reconstructs the input connected to output 56 from the connection memory.

```
unsigned int mcrValue;
unsigned int connValue;
mcrValue = fmic_read(dev0, FMIC_MASTER_REG_ID);
connValue = fmic_read(dev0, FMIC_CONN_MEM_LOW_ID | 56);
connValue |= (fmic_read(dev0, FMIC_CONN_MEM_HIGH_ID | 56))
<<  8) & 0x100;
```

fmic_timing — *Establish Requested Timing Mode*

Syntax

```
#include ”fmic.h”
            int fmic_timing(
            FMIC_DEV_T        fmic_dev,
            fmic_timing_mode timing_mode );
```

Defined in

fmic.c as a callable C routine

Description

This function establishes clock control functionality for the indicated FMIC device.

❑ Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.

❑ Timing_mode selects which clock source is used to generate MVIP clock signals. It can be one of the following values.

- FMIC_MVIP_SLAVE selects the FMIC as timing slave to the MVIP bus.
- FMIC_T1E1_MVIP_MASTER selects the FMIC as timing master.
- FMIC_8KMASTER_MVIP_SLAVE selects the FMIC as timing slave to the MVIP bus.
- FMIC_8KSLAVE_MVIP_MASTER selects the FMIC as timing master.
- FMIC_T1E1_8K_MVIP_MASTER selects the FMIC as timing master.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

The following example sets the timing mode of the FMIC device dev0 to FMIC_MVIP_SLAVE.

```
int status;

status = fmic_timing(dev0, FMIC_MVIP_SLAVE);
if (status == ERROR)
    return(ERROR);
```

fmic_write — *Writes a Value to the Indicated FMIC Register/Memory Location*

Syntax

```
#include ”fmic.h”
int fmic_write(
          FMIC_DEV_T        fmic_dev,
          unsigned int      register_id,
          unsigned int      value,
          unsigned int      mask);
```

Defined in

fmic.c as a callable C routine

Description

This function writes a value to the indicated register/memory location on the specified FMIC device. The write will only affect bit values of the indicated register/memory location set in the mask.

- Fmic_dev is a pointer to a structure containing addressing and state information for the FMIC device.
- Register_id is an identifier for either a direct or indirect register/memory location on the FMIC device. For an indirect location, the following indirect IDs should be OR’ed with the desired location in the register/memory space indicated to obtain the value to pass in as register_id. For example, to obtain the low byte of the 4th element in the connection memory, register_id = FMIC_CONN_MEM_LOW_ID **|** 3. A direct register does not need an offset and can be used as is. The possible register/memory IDs are defined as follows.
    - FMIC_MASTER_REG_ID selects the master control/status register (direct).
    - FMIC_CNTRL_REG_ID selects the FMIC control registers (indirect).
    - FMIC_DATA_MEMORY_ID selects the data memory (indirect).
    - FMIC_CONN_MEM_LOW_ID selects the low byte of the connection memory (indirect).
    - FMIC_CONN_MEM_HIGH_ID selects the high byte of the connection memory (indirect).
    - Value indicates what should be written to the indicated register.
    - Mask specifies the bits that the write will affect. Zero valued bits in the mask will not affect corresponding bits in the register/memory location.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example The following example sets the RESET bit in the master Control/Status register of FMIC device dev0 without affecting any other bits.

```
int status;

status = fmic_write(dev0, FMIC_MASTER_REG_ID, 0x1, 0x1);
if (status == ERROR)
   return(ERROR);
```

3.5 Board Support Library API

This section discusses the McEVM board support library API. Included in this discussion are the functions defined that comprise the board library for the ’C62x McEVM board. No public macros are defined for this module.

The following alphabetical listing includes all of the board support library API functions. Use this listing as a table of contents to the board support library API functions.

Function	Description	Page
cpu_freq	Return current CPU frequency in MHz	3-27
delay_msec	Delay CPU for specified number of milliseconds	3-39
delay_usec	Delay CPU for specified number of microseconds	3-39
evm_codec_disable	Disconnect codec from ’C62x McBSP	3-40
evm_codec_enable	Connect codec to ’C62x McBSP	3-40
evm_default_emif_init	Initialize EMIF for McEVM board to default parameters	3-40
evm_emif_int	Initialize EMIF for McEVM board to clock rate-tailored values	3-41
evm_init	Initialize McEVM board	3-41
evm_led_disable	Disable selected McEVM LED	3-42
evm_led_enable	Enable selected McEVM LED	3-42
evm_nmi_disable	Externally disable NMI	3-43
evm_nmi_enable	Externally enable NMI	3-43
evm_nmi_sel	Select the host or codec as source for NMI	3-43

cpu_freq — *Return Current CPU Frequency in MHz*

Syntax

#include <board.h>
int **cpu_freq(** void **);**

Defined in board.c as a callable C routine

Description The cpu_freq() function determines the internal CPU clock frequency by reading the CLKMODE and CLKSEL bits in the DSPOPT register of the onboard complex programmable logic device (CPLD).

Return Value CPU frequency in Mhz

delay_msec — *Delay CPU for Indicated Number of Milliseconds*

Syntax

#include <board.h>
int **delay_msec(** unsigned short *numMsec* **);**

Defined in board.c as a callable C routine

Description The delay_msec() function uses an available timer to delay the specified number of milliseconds before returning to the caller.

❑ The *numMsec* parameter is a value between 0 and 65 535. For delay values less than 1 ms, use the delay_usec() function.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

delay_usec — *Delay CPU for Indicated Number of Microseconds*

Syntax

#include <board.h>
int **delay_usec(** unsigned short *numUsec* **);**

Defined in board.c as a callable C routine

Description The delay_usec() function uses an available timer to delay the specified number of microseconds before returning to the caller.

❑ The *numUsec* parameter is a value between 0 and 65 535. For delay intervals greater than 65.5 milliseconds, use the delay_msec() function.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_codec_disable

Disconnect codec from ’C62x McBSP

Syntax

#include (board.h>
int **evm_codec_disable(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_codec_disable(() function disconnects the codec from the ’C62x McBSP0 serial port.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_codec_enable

Connect codec to ’C62x McBSP

Syntax

#include (board.h>
int **evm_codec_enable(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_codec_enable() function disconnects the codec from the ’C62x McBSP0 serial port.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_default_emif_init

Initialize EMIF for McEVM Board to Default Parameters

Syntax

#include <board.h>
void **evm_default_emif_init(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_default_emif_init() function initializes the EMIF to default values that will work at any available CPU frequency.

Return Value

None

evm_emif_init *Initialize EMIF for McEVM Board*

Syntax

#include <board.h>
void **evm_emif_init(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_emif_init() function initializes the EMIF to clock rate-tailored values.

Return Value

None

evm_init *Initialize McEVM Board*

Syntax

#include <board.h>
int **evm_init(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_init() function initializes the McEVM board for use by configuring base address variables based upon the ’C62x memory map (MAP 0 or MAP 1) and endian mode selected, initializing the EMIF and enabling the NMI bit in the interrupt enable register. This function must be called before any other DSP support software routines.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

Example

This example initializes the McEVM board for use.

```
int  status;

status = evm_init( );
if (status == ERROR)
  return(ERROR);
```

evm_led_disable *Disable Selected McEVM LED*

Syntax

#include <board.h>
int **evm_led_disable(** int *ledNumber* **);**

Defined in

board.c as a callable C routine

Description

The evm_led_disable() function extinguishes the selected LED on the McEVM board.

❑ Parameter *ledNumber* selects the LED to extinguish. Valid values are 0 (for LED0) or 1 (for LED1).

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_led_enable *Enable Selected McEVM LED*

Syntax

#include <board.h>
int **evm_led_enable(** int *ledNumber* **);**

Defined in

board.c as a callable C routine

Description

The evm_led_enable() function illuminates the selected LED on the McEVM board.

❑ The *ledNumber* parameter selects the LED to illuminate. Valid values are 0 (for LED0) or 1 (for LED1).

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

evm_nmi_disable *Externally Disable NMI*

Syntax

#include <board.h>
void **evm_nmi_disable(**void**);**

Defined in

board.c as a callable C routine

Description

The evm_nmi_disable() function externally disables the NMI source to the ’C62x by clearing the NMIEN bit in the CNTL register of the McEVM CPLD.

Return Value

None

evm_nmi_enable *Externally Enable NMI*

Syntax

#include <board.h>
void **evm_nmi_enable(** void **);**

Defined in

board.c as a callable C routine

Description

The evm_nmi_enable() function externally enables the NMI source to the ’C62x by setting the NMIEN bit in the CNTL register of the McEVM CPLD.

Return Value

None

evm_nmi_sel *Select the Source for NMI*

Syntax

#include <board.h>
void **evm_nmi_sel(** int sel **);**

Defined in

board.c as a callable C routine

Description

The evm_nmi_sel() function selects the source for NMI to the ’C62x by setting or clearing the NMISEL bit in the CNTL register of the McEVM CPLD. Parameter sel selects the host (0) or the codec (1) as the NMI source.

Return Value

None

3.6 T1/E1 Framer Driver Library

The Siemens PEB 2255 T1/E1 Framer and Line Interface supports a single channel bi-directional T1/E1 connection and a single bi-directional 2Mbit serial stream. The interface presented is specific to the use of this chip on the McEvm and potential daughterboard.

3.6.1 T1/E1 API Data Structures and ENUMs

```
typedef  struct{   FALC_REG_MAP*  pxReg;
/* Pointer to FALC registers */
   unsigned char* pu8RcData;
/* Current receive data pointer */

   unsigned char* pu8TxData;
/* Current transmit data pointer */

   unsigned short    u16TxCnt;
/* Transmit byte counter */

   falc_pcm_mode  pcmMode;
/* 0: PCM 30 mode, 1: PCM 24 mode */

   int    l1State;
/* Layer 1 state */

unsigned char dummy [4];
/* Forces size of data structure to */

/* 2^n, which guarantees SHLn */
/* operations instead of IMUL for a */
/* access to structure elements */

}

FALC_DEVICE_CTRL;

typedef struct{ t1e1_state    state;  unsigned int
            base_addr;  FALC_DEVICE_CTRL falc;}T1E1_DEV;

typedef enum{  PCM_30_MODE = 0x00,
            PCM_24_MODE = 0x01}falc_pcm_mode;

typedef enum{  SLAVE_MODE = 0x00,
              MASTER_MODE = 0x01

}

falc_mode;

typedef enum

{

   LOCAL_LOOP_BACK   = 0x01,

   PAYLOAD_LOOP_BACK = 0x02,

   REMOTE_LOOP_BACK  = 0x03,

   CHANNEL_LOOP_BACK = 0x04
}

falc_loopback_mode;

typedef enum

{

   FALC_F12 = 0x00,
   FALC_F4,
   FALC_ESF,
   FALC_F72,
   FALC_DOUBLEFRAME,
   FALC_CRC4_MULTIFRAME,
   FALC_MODIFIED_CRC4_MF

}

falc_frame_mode;

typedef enum

{

   FALC_T1_LINE_CODE = 0,
   FALC_T1_AMI
   FALC_T1_B8ZS,
   FALC_E1_LINE_CODE,
   FALC_E1_AMI,
   FALC_E1_HDB3,
   FALC_LINE_CODE_END

}

falc_line_code;

typedef enum{ T1E1_E1 = PCM_30_MODE,
   T1E1_T1 =    PCM_24_MODE}
t1e1_line_mode;
typedef enum{  T1E1_T1_AMI  =
   FALC_T1_AMI,  T1E1_T1_B8ZS =
   FALC_T1_B8ZS,  T1E1_E1_AMI  = FALC_E1_AMI,
   T1E1_E1_HDB3 = FALC_E1_HDB3

}

t1e1_line_code;

typedef enum

{

   T1E1_T1_D4                = FALC_F12,

   T1E1_T1_ESF               = FALC_ESF,

   T1E1_E1_DOUBLEFRAME       = FALC_DOUBLEFRAME,

   T1E1_E1_CRC4_MULTIFRAME  = FALC_CRC4_MULTIFRAME

}

t1e1_framing_mode;
```

```
typedef enum
{             /* LIM0.MAS LIM1.DCOC FMICMSTR */
   T1E1_SYNC_TIMING = 0, /* 1    1     1     */
   T1E1_LOOP_TIMING,     /* 0    1     x     */
   T1E1_LOCAL_TIMING     /* 1    1     0     */
}
t1e1_sync_mode;
typedef enum
{
   T1E1_NO_LOOPBACK = 0,
   T1E1_LOCAL_LOOPBACK = LOCAL_LOOP_BACK,
   T1E1_PAYLOAD_LOOPBACK = PAYLOAD_LOOP_BACK,
   T1E1_REMOTE_LOOPBACK = REMOTE_LOOP_BACK,
   T1E1_CHANNEL_LOOPBACK = CHANNEL_LOOP_BACK,
   T1E1_LOCAL_REMOTE_LOOPBACK
}
t1e1_loopback_mode;
```

3.6.2 Siemens PEB 2255 T1/E1 Framer and Line Interface Registers

Register information is given in Siemens PEB 2255 data sheet T2255-XV11-P1-7600.

3.6.3 T1/E1 API Functions

This section includes the library T1/E1 Framer Driver API functions for the C6x McEVM. The following is an alphabetical listing of these API functions which can be used as a table of contents.

Function	Description	Page
t1e1_channel_loopback	Enable or disable channel loopback for a specified channel.	3-48
t1e1_close	De-initialize the software driver for the T1/E1 device.	3-49
t1e1_configure	Register initialization from a configuration array of registers and values.	3-49
t1e1_framing	Set the Framing Mode register.	3-50
t1e1_init	Initialize the T1/E1 framer to specified values.	3-50
t1e1_install	Install a callback function to handle interrupts.	3-51

Function	Description	Page
t1e1_linecode	Set the RX and TX Line Code register.	3-52
t1e1_loopback	Enable or disable Payload, Remote, and/or Local Loopbacks.	3-52
t1e1_open	Initialize the software driver for the T1/E1 device.	3-53
t1e1_read	Read the specified T1/E1 register.	3-53
t1e1_reset	Reset all register values to power on states.	3-54
t1e1_sync	Set the sync timing mode for the T1/E1 Framer.	3-54
t1e1_write	Write to the specified T1/E1 register.	3-55

t1e1_channel_loopback

Enables or Disables Loopback for Specified Channel

Syntax

```
#include<t1e1.h>
int t1e1_channel_loopback(
            T1E1_DEV_T      dev,
            unsigned int    channel,
            unsigned int    channel_loopback_enable,
            unsigned int    idle_code)
```

Defined in

t1e1.c as a callable C routine

Description

The t1e1_channel_loopback function either enables or disables loopback for a specified channel.

- The *t1e1_dev* parameter refers to the initialized device handle returned by t1e1_open.
- The *channel* parameter specifies the channel to loopback.
- The *channel_loopback_enable* parameter values are1=enable, 0=disable.
- The *idle_code* is the value for idle code insertion.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_close

De-initializes the Software Driver

Syntax

```
#include<t1e1.h>
int t1e1_close(
            T1E1_DEV_T      dev)
```

Defined in

t1e1.c as a callable C routine

Description

The t1e1_close function invalidates the T1/E1 device handle.

❑ The parameter *t1e1_dev* refers to the initialized device handle returned bt t1e1_open.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_configure

Allows Register Initialization

Syntax

```
#include<t1e1.h>
int t1e1_configure(
            T1E1_DEV_T      t1e1dev,
                            t1e1_configuration)
```

Defined in

t1e1.c as a callable C routine

Description

This function allows register initialization via a configuration array.

❑ The *t1e1_dev* parameter refers to the initialized device handle returned by t1e1_open.

❑ The *t1e1_configuration* parameter is an array of tuples of the form (unsigned int register_id, unsigned int value, unsigned int mask) and is terminated by an entry with a mask of 0.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_framing

Sets the Framing Mode Register

Syntax

```
#include<t1e1.h>
int t1e1_framing(
        T1E1_DEV_T      t1e1_dev,
        unsigned int    frame_mode)
```

Defined in

t1e1.c as a callable C routine

Description

Sets the framing mode register according to value specified in frame_mode.

❑ The *t1e1_dev* parameter refers to the initialized device handle returned by t1e1_open.

❑ For a description of the *frame_mode* parameter see *t1e1_framing_mode* enum list.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_init

Initializes the T1/E1 Framer to Specified Values

Syntax

```
#include<t1e1.h>
int  t1e1_init(
        T1E1_DEV_T      t1e1_dev,
        unsigned int    line_mode
        unsigned int    line_code_mode,
        unsigned int    framing_mode,
        unsigned int    sync_mode,
        unsigned int    loopback_mode)
```

Defined in

t1e1.c

Description

Establish normal operating register values for the T1/E1 Framer. These include setting line mode (T1 or E1), line code mode (AMI and B8ZS or AMI and HDB3), framing mode (D4, ESF **|** CRC_4_MULTIFRAME, DOUBLE_FRAME), sync mode (loop , FMIC **|** loop, FMIC), and loopback mode(s) (none, local, payload, remote, local and remote).

❑ The *t1e1_dev* parameter refers to the initialized device handle returned by t1e1_open.

- ❑ For a description of the *line_mod*e parameter see t1e1_line_mode enum list.
- ❑ For a description of the *line_code_mode* parameter see t1e1_line_code_mode enum list.
- ❑ For a description of the *framing_mode* parameter see t1e1_framing_mode enum list.
- ❑ For a description of the *sync_mode* parameter see t1e1_sync_mode enum list.
- ❑ For a description of the *loopback_mode* parameter see t1e1_loopback_mode enum list.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_install — *Installs a Callback Function to Handle Interrupts*

Syntax

```
#include<t1e1.h>
int t1e1_install(
            T1E1_DEV_T        t1e1_dev,
            unsigned int      register_id,
            unsigned int      mask,
            T1E1_CALLBACK_T   *pf_ract)
```

Defined in

t1e1.c as a callable C routine

Description

The parameters are as follows:

- ❑ The t1e1_dev parameter refers to the initialized device handle returned by t1e1_open.
- ❑ The register_id parameter refers to the interrupt mask register (0x14-0x19).
- ❑ The mask parameter refers to the interrupt mask (bit = 1 sets interrupt active).
- ❑ The *pf_ract parameter refers to the function which interrupt service routines will call.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_linecode — *Sets the RX and TX Line Code Registers*

Syntax

```
#include<t1e1.h>
int t1e1_linecode(
            T1E1_DEV_T      t1e1_dev,
            unsigned int    line_code_mode)
```

Defined in t1e1.c as a callable C routine

Description The t1e1_linecode function sets the RX and Tx Line Code Registers.

- ❑ The t1e1_dev parameter refers to the initialized device handle returned by t1e1_open.
- ❑ For a description of the line_code_mode parameter see t1e1_line_code_mode enum list.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_loopback — *Enables or Disables Payload, Remote, and/or Local Loopback(s)*

Syntax

```
#include<t1e1.h>
int t1e1_loopback(
            T1E1_DEV_T      t1e1_dev,
            unsigned int    loopback_mode)
```

Defined in t1e1.c as a callable C routine

Description The t1e1_loopback function enable or disable the specified loopback.

- ❑ The t1e1_dev parameter refers to the initialized device handle returned by t1e1_open.
- ❑ For a description of the loopback_mode, see t1e1_loopback_mode enum list. (T1E1_NO_LOOPBACK to disable).

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_open

Initializes the Software Driver

Syntax

```
#include<t1e1.h>
T1E1_DEV_T t1e1_open(
            unsigned int       t1e1_base_address)
```

Defined in

t1e1.c as a callable C routine

Description

Provides a handle to reference a T1/E1 device.

❑ The parameter *t1e1_base_address* is the base address of the T1/E1 device.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_read

Returns the Value Contained in the Indicated Register

Syntax

```
#include<t1e1.h>
unsigned int t1e1_read(
            T1E1_DEV_T         t1e1_dev,
            unsigned int       register_id)
```

Defined in

t1e1.c as a callable C routine

Description

The t1e1_read function reads the specified register.

❑ The *t1e1_dev* parameter refers to the initialized device handle returned by t1e1_open.

❑ The *register_id* parameter refers to the register number.

Return Value

Value of specified register.

t1e1_reset — *Resets All Register Values to Power-Up States*

Syntax

```
#include<t1e1.h>
int t1e1_reset(
              T1E1_DEV_T      t1e1_dev)
```

Defined in

t1e1.c as a callable C routine

Description

Reset all registers values to power up status.

❑ The parameter t1e1_dev refers to the initialized device handle returned by t1e1_open.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_sync — *Sets the Sync Timing Mode for the T1/E1 Framer*

Syntax

```
#include<t1e1.h>
int t1e1_sync(
              T1E1_DEV_T      t1e1_dev,
              unsigned int    sync_mode)
```

Defined in

t1e1.c as a callable C routine

Description

Set the sync timing mode for the T1/E1 Framer.

❑ The t1e1_dev parameter refers to the initialized device handle returned by t1e1_open.

❑ For a description of the sync_mode parameter see t1e1_sync_mode enum list.

Return Value

The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

t1e1_write — *Writes to the specified T1/E1 Register*

Syntax

```
#include<t1e1.h>
int t1e1_write(
          T1E1_DEV_T      t1e1_dev,
          unsigned int    register_id,
          unsigned int    value,
          unsigned int    mask)
```

Defined in t1e1.c as a callable C routine

Description Write a value into a register, limit the bits written to those which are set in mask.

Return Value The function returns one of the following values:

OK	Operation succeeded
ERROR	Operation failed

3.7 VBAP Driver Library API

The TCM320AC36/7 voice-band audio processor (VBAP) integrated circuits are designed to perform the transmit and receive encoding and decoding (A/D and D/A conversions) together with transmit and receive filtering for voice-band communications systems. The interface presented is specific to the use of this chip on the McEVM.

3.7.1 VBAP API ENUMs

```
typedef enum { VBAP_A_LAW=0, VBAP_U_LAW}vbap_law;
```

3.7.2 VBAP Library API Functions

The following alphabetical listing includes all of the VBAP library API functions. Use this listing as a table of contents to the VBAP library API functions.

vbap_set — *Sets the Companding type for A/D and D/A Conversions*

Syntax

```
#include<vbap.h>
int vbap_set(vbap_law law);
```

Defined in In vbap.c as a callable C routine

Description Sets the companding type to that specified (A-law or µ-law). See vbap_law enum list for parameter law.

Return Value Returns OK

vbap_get — *Gets the Current Companding Type for A/D and D/A Conversions*

Syntax

```
#include<vbap.h>
vbap_law vbap_get(void);
```

Defined in In vbap.c as a callable C routine

Description Gets the current companding type.

Return Value Returns current companding selection.(See vbap_law enum list).OK

3.8 PCI/AMCC Driver Library API

This section discusses the PCI/AMCC library API. Included in this discussion are the macros, data types, and defined functions that comprise the PCI/AMCC library for the ’C62x McEVM board.

3.8.1 PCI/AMCC Library Data Types and Macros

```
/* function typedefs for callback functions*
/typedef void pci_fifo_callback( int  status );
 typedef void pci_msg_callback( int  status );
```

3.8.2 PCI/AMCC Library API Functions

The following alphabetical listing includes all of the PCI/AMCC library API functions. Use this listing as a table of contents to the PCI/AMCC library API functions.

Function	Description	Page
pci_driver_int(void)	Initializes the PCI Driver	3-59
pci_fifo_open	Opens the driver for the PCI FIFO device	3-60
pci_fifo_close	Closes the driver for the PCI FIFO device	3-61
pci_fifo_async_send	Starts an asynchronous PCI FIFO send operation	3-62
pci_fifo_sync_send	Starts a synchronous PCI FIFO send operation	3-63
pci_fifo_async_receive	Starts an asynchronous PCI FIFO receive operation	3-64
pci_fifo_sync_receive	Starts a synchronous PCI FIFO receive operation	3-65
pci_message_send	Sends a message immediately	3-66
pci_message_async_send	Starts an asynchronous message operation	3-67
pci_message_sync_send	Starts a synchronous message operation	3-68
pci_message_retrieve	Retrieves a message immediately	3-69
pci_message_async_retrieve	Starts an asynchronous retrieve message operation	3-70
pci_message_sync_retrieve	Starts a synchronous retrieve message operation	3-71
amcc_nvram_read	Reads a byte from NVRAM	3-72
amcc_nvram_write	Writes a byte from NVRAM	3-73
amcc_mailbox_read	Reads a AMCC mailbox	3-74
amcc_mailbox_write	Writes to a AMCC mailbox	3-75

pci_driver_init(void)

Initializes the PCI Driver

Syntax

#include <pci.h>
void **pci_driver_init**(void)

Defined in

pci.c as a callable C routine

Description

This routine initializes the software driver if it has not been initialized previously. This includes setting up the messaging and the PCI FIFO transfer portions of the driver. Note that this function must be called before using any of the driver calls. But, also note that this function is called as part of the pci_fifo_open() function. So, if the pci_fifo_open() call is the first use of this driver, the pci_driver_init() is not required.

Also, the messaging calls in this driver use the mailbox 1 register of the AMCC PCI controller in both directions. Thus, this mailbox is not available for direct accesses using the mailbox read and write routines.

Return Value

None

Example

```
#include <board.h>
#include <pci.h>

main()
{
    evm_init();
    pci_driver_init();
. . .

}
```

pci_fifo_open *Opens the Driver for the PCI FIFO Device*

Syntax

#include <pci.h>
int **pci_fifo_open**(void)

Defined in

pci.c as a callable C routine

Description

This routine opens and initializes the PCI FIFO device used to communicate with the host.

Return Value

return a channel number to reference the device or an ‘invalid’ channel number if the device is already open (an invalid channel number is anything less than 0)

Example

```
#include <board.h>
#include <pci.h>
. . .

    int  pci_chan;

    evm_init();
    pci_chan = pci_fifo_open();

    if ( pci_chan < 0 )
    {
        /* pci_fifo_open() failed */
    }

. . .
```

pci_fifo_close — *Closes the Driver for the PCI FIFO Device*

Syntax

#include <pci.h>
int **pci_fifo_close**(int *chan*)

Defined in

pci.c as a callable C routine

Description

This routine closes the PCI FIFO device.

- The chan parameter is the channel number returned from a successful open.

Return Value

Returns OK or ERROR; possible error conditions include: device not open

Example

```
#include <board.h>
#include <pci.h>
. . .

    int  pci_chan;

    evm_init();
    pci_chan = pci_fifo_open();

. . .

    pci_fifo_close( pci_chan );

. . .
```

pci_fifo_async_send

Starts an Asynchronous PCI FIFO Send Operation

Syntax

```
#include <pci.h>
int pci_fifo_async_send(
        int                  chan,
        unsigned int         *p_buffer,
        unsigned int         num_bytes,
        pci_host_callback *p_callback )
```

Defined in

pci.c as a callable C routine

Description

This routine starts a data transfer to the host using DMA. It returns immediately, then the callback function is executed when the operation has completed. Note that each individual bus transaction is 32bits in size so num_bytes must be a multiple of four and p_buffer must be 32bit aligned. If a close is called during a pending async operation, the operation is aborted and the callback is called with an error status. Also, only one FIFO send operation is supported at a time, but it can be concurrent with a FIFO receive operation.

❑ The chan parameter is the channel number from a successful open.

❑ The p_buffer parameter is the location of the data to be written to the FIFO. This address must be 32-bit aligned.

❑ The num_bytes parameter is the number of bytes of data to be written to the FIFO. This number must be a multiple of 4.

❑ The p_callback parameter is the function pointer for the callback routine.

Return Value

OK or ERROR
Possible error conditions include: another send in progress, chan not open, no DMA chan available.

Example

```
#include <board.h>
#include <pci.h>
. . .
    int  pci_chan;
    evm_init();
    pci_chan = pci_fifo_open();
. . .
```

pci_fifo_sync_send

Starts a Synchronous PCI FIFO Send Operation

Syntax

#include <pci.h>
int **pci_fifo_sync_send(**

int	*chan,*
unsigned int	**p_buffer,*
unsigned int	*num_bytes* **)**

Defined in

pci.c as a callable C routine

Description

This routine sends data to the PCI FIFO. It will return when the transfer is complete. This function does not use DMA; it polls the FIFO flags and writes data to the FIFO directly.

- The chan parameter is the channel number from a successful open.
- The p_buffer parameter is the location of the data to be written to the FIFO. This address must be 32 bit aligned
- The num_bytes parameter is the number of bytes of data to be written to the FIFO. This number must be a multiple of 4

Return Value

OK or ERROR
Possible error conditions include: chan not open, another send in progress.

Example

```
#include <board.h>
#include <pci.h>
unsigned int     buffer[0x48];
. . .

    int  pci_chan;

    evm_init();
    pci_chan = pci_fifo_open();

    pci_fifo_sync_send( pci_chan, buffer, 0x48*4 );
. . .
```

pci_fifo_async_receive

Starts an Asynchronous PCI FIFO Receive Operation

Syntax

#include <pci.h>
int **pci_fifo_async_receive(**

int	*chan,*
unsigned int	**p_buffer,*
unsigned int	*num_bytes,*
pci_host_callback	**p_callback* **)**

Defined in

pci.c as a callable C routine

Description

This routine begins an asynchronous FIFO receive operation using DMA. It returns immediately and the callback function is called when the operation completes. Note that each individual bus transaction is 32 bits in size so num_bytes must be a multiple of four and p_buffer must be 32bit aligned. If a close is called during a pending async operation, the operation is aborted and the callback is called with an error status. Also, only one FIFO receive operation is supported at a time, but it can be concurrent with a FIFO send operation.

❑ The chan parameter is the channel number from a successful open.

❑ The p_buffer parameter is the location of the data to be written to the FIFO. This address must be 32-bit aligned.

❑ The num_bytes parameter is the number of bytes of data to be written to the FIFO. This number must be a multiple of 4.

❑ The p_callback parameter is the function pointer for the callback routine.

Return Value

OK or ERROR.
Possible error conditions include: chan not open, another receive in progress, no DMA chan available.

Example

```
#include <board.h>
#include <pci.h>
. . .
    int  pci_chan;
    evm_init();
    pci_chan = pci_fifo_open();
. . .
```

pci_fifo_sync_ receive

Starts a Synchronous PCI FIFO Receive Operation

Syntax

#include <pci.h>
int **pci_fifo_sync_receive(**
int *chan,*
unsigned int **p_buffer,*
unsigned int *num_bytes* **)**

Defined in

defined in pci.c as a callable C routine

Description

This routine receives data from the PCI FIFO. It will return when the transfer is complete. This function does not use DMA; it polls the FIFO flags and reads data from the FIFO directly.

- ❑ The chan parameter is the channel number from a successful open.
- ❑ The p_buffer parameter is the location of the data to be written to the FIFO. This address must be 32 bit aligned.
- ❑ The num_bytes parameter is the number of bytes of data to be written to the FIFO. This number must be a multiple of 4.

Return Value

OK or ERROR
Possible error conditions include: chan not open, another receive in progress.

Example

```
#include <board.h>
#include <pci.h>
unsigned int     buffer[72];
. . .
    int  pci_chan;

    evm_init();
    pci_chan = pci_fifo_open();

    pci_fifo_sync_receive( pci_chan, buffer, 72<<2 );
. . .
```

pci_message_send

Sends a Message Immediately

Syntax

#include <pci.h>
int **pci_message_send**(unsigned int *message*)

Defined in

pci.c as a callable C routine

Description

This routine sends a 32-bit message. If the message cannot be placed into the mailbox immediately, it returns with an error and the message is not sent. This routine checks mailbox empty/full flags and the HINT bit. If outgoing mailbox 1 is empty and HINT is clear, it places message into mailbox 1. It then sets the HINT bit to cause an interrupt to the host.

❑ The message parameter is the 32-bit value to be sent to the host.

Return Value

OK or ERROR
Possible error conditions include: outgoing message mailbox not empty, HINT not clear.

Example

```
#include <board.h>
#include <pci.h>
. . .

    evm_init();
    pci_driver_init();
. . .
    if ( pci_message_send( 0x1234fedc ) != OK )
    {
        /* message send failed */
    }
. . .
```

pci_mesage_ async_send

Starts an Asynchronous Message Operation

Syntax

#include <pci.h>
int **pci_message_async_send(**
unsigned int *message,*
bool *wait_for_ack,*
pci_message_callback **p_callback* **)**

Defined in

pci.c as a callable C routine

Description

This routine will begin a send message operation. The call will return immediately. The callback will be called when the operation is complete. This routine uses interrupts internally; it does not poll. If wait_for_ack is TRUE, then callback is not called until the message has been read by the host. If it is FALSE then callback is called as soon as the message is placed into the outgoing mailbox.

- The message parameter is the 32-bit value to be sent to the host.
- The wait_for_ack parameter determines when the operation is considered complete and the callback function is called.
- The p_callback parameter is the funciton pointer for the callback routine.

Return Value

OK or ERROR
Possible error conditions include: another send in progress.

Example

```
#include <board.h>
#include <pci.h>
. . .
    int  pci_chan;

    evm_init();
    pci_chan = pci_fifo_open();
. . .
```

pci_message_sync_send

Starts a Synchronous Message Operation

Syntax #include <pci.h>

Syntax int **pci_message_sync_send**(unsigned int message, bool wait_for_ack)

Defined in pci.c as a callable C routine

Description This routine will send a 32bit message to the host. It will not return until the operation is complete. If wait_for_ack is TRUE, then this function is not complete until the message has been read by the host. If it is FALSE, then this function is complete as soon as the message is placed into the outgoing mailbox. This function is implemented by calling pci_message_async_send() and waiting internally for the operation to complete.

❑ The message parameter is the 32-bit value to be sent to the host.

❑ The wait_for_ack parameter determines when the operation is considered complete.

Return Value OK or ERROR
Possible error conditions include: another send in progress.

Example

```
#include <board.h>
#include <pci.h>
. . .
    int  pci_chan;


    evm_init();
    pci_driver_init();
        /* send message, return when message is read by
host */
    pci_message_sync_send( 0x00223300, TRUE );
. . .
```

pci_message_ retrieve

Retrieves a Message Immediately

Syntax

#include <pci.h>
int **pci_message_retrieve**(unsigned int **p_message*)

Defined in

pci.c as a callable C routine

Description

This routine retrieves a message sent by the host. If there is no message available, it returns ERROR. This function checks mailbox empty/full flags; if mailbox 1 is full, the message is read from the mailbox 1 register.

❑ The p_message parameter is the location to store the 32-bit value retrieved from to the host.

Return Value

OK or ERROR
Possible error conditions include: incoming message mailbox not full.

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned int    *p_message;

    evm_init();
    pci_driver_init();
    if ( pci_message_retrieve( p_message ) != OK )
    {
        /* no message available */
    }
. . .
```

pci_message_ async_retrieve

Starts an Asynchronous Retrieve Message Operation

Syntax

#include <pci.h>

Syntax

int **pci_message_async_retrieve(**
unsigned int **p_message,*
pci_message_callback **p_callback* **)**

Defined in

pci.c as a callable C routine

Description

This routine begins an asynchronous message retrieve operation. It returns immediately. When the operation is complete, the callback function is called. This function uses interrupts internally; it does not poll.

- ❑ The p_message parameter is the location to store the 32-bit value retrieved from to the host.
- ❑ The p_callback parameter is the function pointer for the callback routine.

Return Value

OK or ERROR
Possible error conditions include: another retrieve in progress.

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned int  *p_message;

    evm_init();
    pci_driver_init();
. . .
```

pci_message_sync_retrieve

Starts a Synchronous Retrieve Message Operation

Syntax #include <pci.h>

Syntax int **pci_message_sync_retrieve**(unsigned int *p_message)

Defined in pci.c as a callable C routine

Description This routine will retrieve a 32-bit message sent by the host. It will not return until the operation is complete. This function is implemented by calling pci_message_async_retrieve() and waiting internally for the operation to complete.

❑ The p_message parameter is the location to store the 32-bit value retrieved from the host.

Return Value returns OK or ERROR, possible error conditions include: another retrieve in progress.

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned int  *p_message;

    evm_init();
    pci_driver_init();
    pci_message_sync_retrieve( p_message );
        /* retrieved message in *p_message */
. . .
```

amcc_nvram_read

Reads a Byte from NVRAM

Syntax

#include <pci.h>
int **amcc_nvram_read**(
unsigned short *offset,*
unsigned char **p_data*)

Defined in

pci.c as a callable C routine

Description

This routine returns the byte of data at the indicated offset to the p_data address. Note that the NVRAM device on the EVM and McEVM boards is 2K bytes in size.

❑ The offset parameter is the address offset into the NVRAM device.

❑ The p_data parameter is the location to store the 8-bit value read from NVRAM.

Return Value

OK or ERROR
Possible error conditions include: offset out of range.

Example

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned char   uc_data;

    evm_init();
    pci_driver_init();
    amcc_nvram_read( 0x01e8, &uc_data );
. . .
```

amcc_nvram_write

Writes a Byte to NVRAM

Syntax

```
#include <pci.h>
int amcc_nvram_write(
        unsigned short    offset,
        unsigned char     data )
```

Defined in

pci.c as a callable C routine

Description

This routine will write the byte of data to NVRAM at the indicated offset. This routine will not allow the modification of data between offsets of 0x0000 and 0x007f. Offsets 0x0000 through 0x007f are used for PCI configuration and should not be modified.

- ❑ The offset parameter is the address offset into the NVRAM device.
- ❑ The data parameter is the 8-bit value to write to NVRAM.

Return Value

OK or ERROR, possible error conditions include: invalid offset, offset out of range.

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned char           uc_data;

    evm_init();
    uc_data = 0x4e;

    pci_driver_init();
    amcc_nvram_write( 0x02f0, uc_data );
. . .
```

amcc_mailbox_read

Reads a AMCC Mailbox

Syntax

#include <pci.h>
int **amcc_mailbox_read(**
    int *mailbox_number,*
    unsigned int **p_data*)

Defined in

pci.c as a callable C routine

Description

This routine will read the contents of one of the mailbox registers not used by the pci_message_xxx() calls. If the indicated mailbox is not full this routine will return ERROR. Note that mailbox register 1 is used for the pci_message_xxx() calls so it is not available to this routine.

❑ The mailbox_number parameter is the mailbox register number to be checked. It must be a value of 2, 3 or 4.

❑ The p_data parameter is the location to store the 32-bit value sent by the host.

Return Value

OK or ERROR
Possible error conditions include: invalid mailbox number (must be 2, 3 or 4), mailbox not full.

Example

Example

```
#include <board.h>
#include <pci.h>
. . .
    unsigned int    data;

    evm_init();
    pci_driver_init();
    amcc_mailbox_read( 2, &data );
. . .
```

amcc_mailbox_write

Writes to a AMCC Mailbox

Syntax

#include <pci.h>
int **amcc_mailbox_write**(
int *mailbox_number,*
unsigned int *data* **)**

Defined in

pci.c as a callable C routine

Description

This routine will place a 32-bit word into one of the mailbox registers not used by the pci_mesage_xxx() calls. If the indicated mailbox is not empty, this routine will return ERROR. Mailbox register 1 is used for the pci_message_xxx() calls so it is not available to this routine.

Note that byte 3 of outgoing mailbox register 4 is not writeable so writing to this byte will have no effect. On the host side, byte 3 of incoming mailbox register 4 contains hardware signal information (see the EVM reference guide for details).

❑ The mailbox_number parameter is the mailbox register number to be checked. It must be a value of 2, 3 or 4.

❑ The data parameter is the 32bit value to be sent to the host.

Return Value

OK or ERROR, possible error conditions include: invalid mailbox number (must be 2, 3 or 4), mailbox not empty

Example

Example

```
#include <board.h>
#include <pci.h>
. . .
    evm_init();
    pci_driver_init();
    amcc_nvram_write( 3, 0x9876abcd );
. . .
```

3.9 C I/O Interface Library API

This section discusses the C I/O Interface library API. Included in this discussion are the macros, data types, and defined functions that comprise the C I/O Interface library for the ’C62x McEVM board.

The following alphabetical listing includes all of the C I/O Interface library API functions. Use this listing as a table of contents to the C I/O Interface library API functions.

Function	Description	Page
cio_pci_fifo_open	Opens the C I/O interface to the AMCC/PCI driver	3-77
cio_pci_fifo_read	Reads from the host using the AMCC/PCI driver	3-78
cio_pci_fifo_write	Writes to the host using the AMCC/PCI driver	3-79
cio_pci_fifo_lseek	Sets the file position indicator [not supported]	3-79
cio_pci_fifo_close	Closes the C I/O interface to the AMCC/PCI driver	3-80
cio_pci_fifo_unlink	Deletes the file [not supported]	3-80
cio_pci_fifo_rename	Renames the file [not supported]	3-80

cio_pci_fifo_open

Opens the C I/O Interface to the AMCC/PCI Driver

Syntax

#include <cio_fifo.h>
int **cio_pci_fifo_open(**
const char **path,*
unsigned *flags,*
int *mode* **)**

Defined in cio_fifo.c as a callable C routine

Description This routine opens the FIFO channel to the host. Note that only one stream is supported and that this function utilizes the pci_fifo_open() routine.

- ❑ The path parameter is ignored.
- ❑ The mode parameter is ignored.
- ❑ The flags parameter specify how the device is manipulated.

Return Value A stream number on success or <0 on failure

Example

```
#include <board.h>#include <stdio.h>#include <cio_fifo.h>

unsigned int buffer[0x80];
main()
{
    FILE            *fid;
    evm_init();
    add_device( “pci_fifo”, _SSA,
                    cio_pci_fifo_open,
cio_pci_fifo_close,                 cio_pci_fifo_read,
                    cio_pci_fifo_write,
                    cio_pci_fifo_lseek,
                    cio_pci_fifo_unlink,
                    cio_pci_fifo_rename );

    fid = fopen( “pci_fifo:”, “rw” );
```

```
        /* read from host using pci_fifo device */
fread( buffer, 4, 0x80, fid );


        /* write to host using pci_fifo device */
fwrite( buffer, 4, 0x80, fid );


    fclose( fid );
```

cio_pci_fifo_read

Reads the C I/O Interface to the AMCC/PCI Driver

Syntax #include <cio_fifo.h>

Syntax int **cio_pci_fifo_read(**

int	*fildes,*
char	**bufptr,*
unsigned	*cnt* **)**

Defined in cio_fifo.c as a callable C routine

Description This function reads the number of characters indicated by 'cnt' from the pci_fifo device.

- ❑ The fildes parameter is the stream number returned from a successful open call.
- ❑ The bufptr parameter is the location of the buffer where read data is placed.
- ❑ The cnt parameter is the number of characters to be read from the device.

Return Value Returns the number of characters read or -1 on failure

Example See the example for pci_fifo_open().

cio_pci_fifo_ write

Writes to the Host Using the AMCC/PCI Driver

Syntax

```
#include <cio_fifo.h>
int cio_pci_fifo_write(
            int             fildes,
            const char      *bufptr,
            unsigned        cnt )
```

Defined in cio_fifo.c as a callable C routine

Description This function writes the number of charcters indicated by *cnt* to the pci_fifo device.

- ❑ The fildes parameter is the stream number returned from a successful open call.
- ❑ The bufptr parameter is the location of the buffer of data to be sent to the device.
- ❑ The cnt parameter is the number of characters to be sent to the device.

Return Value Returns the number of characters written or -1 on failure.

Example See the example for pci_fifo_open().

cio_pci_fifo_ lseek

Sets the File Position Indicator

Syntax

```
#include <cio_fifo.h>
int cio_pci_fifo_lseek(
int         fildes
long        offset
int         origin )
```

Defined in cio_fifo.c as a callable C routine

Description This function is not supported by the pci_fifo device.

Return Value Always returns E0F (lseek is not supported).

cio_pci_fifo_close

Closes the C I/O Interface to the AMCC/PCI Driver

Syntax

#include <cio_fifo.h>
int **cio_pci_fifo_close**(int *fildes)*

Defined in

cio_fifo.c as a callable C routine

Description

This function closes the pci_fifo device.

- ❑ The fildes parameter is the stream number returned from a successful open call.

Return Value

Returns 0 on success or -1 on failure.

Example

See the example for pci_fifo_open().

cio_pci_fifo_unlink

Deletes the File

Syntax

#include <cio_fifo.h>
int **cio_pci_fifo_unlink**(const char **path*)

Defined in

cio_fifo.c as a callable C routine

Description

This function is not supported by the pci_fifo device.

Return Value

Always returns -1 (unlink is not supported).

cio_pci_fifo_rename

Renames the File

Syntax

```
#include <cio_fifo.h>
int cio_pci_fifo_rename(
            const char          *old_name,
            const char          *new_name )
```

Defined in

cio_fifo.c as a callable C routine

Description

This function is not supported by the pci_fifo device.

Return Value

Always returns -1 (rename is not supported).

Appendix A

TMS320C62x McEVM Connector Pinouts

This appendix contains the pinout information for each connector on the TMS320C62x McEVM.

Topic **Page**

A.1 TMS320C62x McEVM Connector Summary

There are ten connectors on the ’C62x McEVM, as shown in Table A–1. The J7 CPLD ISP connector is for factory use and is not installed.

Table A–1. TMS320C62x McEVM Connectors Summary

Connector	No. of Pins	Description	Type	See Page
J1	3	Handset microphone in (mono)	3.5-mm audio jack	A-2
J2	3	Handset ear out (mono)	3.5-mm audio jack	A-3
J3	8	T1/E1 interface	RJ-48C modular jack, twisted-pair	A-3
J4	40	MVIP-90 interface	40-pin, right-angle, ribbon cable	A-4
J5	3	MVIP /C2 termination jumper	1 × 3 pin, 0.1 in.	A-6
J6	3	MVIP /C4 termination jumper	1 x 3 pin, 0.1 in.	A-6
J7	10	CPLD ISP header	2 × 5 pin, 0.1 in.	A-7
J8	80	Expansion memory interface	2 x 40 pos., 0.050-in. SMT	A-8
J9	80	Expansion peripheral interface	2 x 40 pos., 0.050-in. SMT	A-9
J10	14	’C62x JTAG emulation header	2 x 7 pin, 0.1 in.	A-10
J11	4	External power	Molex disk driver, right-angle	A-11
J12	2	DSP fan power (not used)	Molex 1.25-mm, right-angle	A-11
P1	124	PCI local bus	Edge connector	A-12

A.2 Microphone Input Jack (Mono)

Connector J1 supports a stereo microphone input. Sjince it is a mono microphone, only the input left (tiup) channel is used.

Table A–2. Stereo Microphone Input Connector J1 Pinout

J1 Pin No.	Signal Name	Description	Type
1	Tip	Mic In	I
2	Ring	NC	–
3	Sleeve	AGND	–

A.3 Handset EarOut Mono

Connector J2 supports a mono earphone output. The line output audio connector is the bottom 3.5-mm jack on the EVM’s mounting bracket.

Table A–3. Handset Output Connector J2 Pinout

J2 Pin No.	Signal Name	Description	Type
1	Tip	Ear out	O
2	Ring	NC	–
3	Sleeve	AGND	–

A.4 RJ-48C T1/E1 Interface Connector

T1/E1 interface J3 connector provides a multichannel, digital telephone interface for the ’C6201 DSP that allows it to process multiple channels. The T1/E1 output is presented as an RJ-48C twisted-pair modular jack on the board’s mounting bracket.

Table A–4. T1/E1 Interface Connector J3 Pinouts

J3 Pin No.	Signal Name	Description	Type
1	RxRing	Receive Ring	I
2	RxTip	Receive Tip	I
3	NC	–	–
4	TxRing	Transmit ring	0
5	TxTip	Transmit tip	0
6	NC	–	–
7	NC	–	–
8	NC	–	–

A.5 MVIP-90 Interface

Connector J4 provides interfaction with a wide rage of third-party telephony boards. The MVIP-90 interface consists of a 40-pin connector located at the top of the McEVM board.

Table A–5. MVIP-90 Interface J4 Connector Pinout

J4 Pin No.	Signal Name	Description	Type
1	Reserved	–	–
2	Reserved	–	–
3	Reserved	–	–
4	Reserved	–	–
5	Reserved	–	–
6	Reserved	–	–
7	DSoO	Data Stream Out 0	O
8	DSiO	Data Stream In 0	I
9	Dso1	Data Stream Out 1	O
10	Dsi1	Data Stream In 1	I
11	Dso2	Data Stream Out 2	O
12	Dsi2	Data Stream In 2	I
13	Dso3	Data Stream Out 3	O
14	Dsi3	Data Stream In 3	I
15	Dso4	Data Stream Out 4	O
16	Dsi4	Data Stream In 4	I
17	Dso5	Data Stream Out 5	O
18	Dsi5	Data Stream In 5	I
19	Dso6	Data Stream Out 6	O
20	Dsi6	Data Stream In 6	I
21	Dso7	Data Stream Out 7	O
22	Dsi7	Data Stream In 7	I
23	Reserved	–	–

J4 Pin No.	Signal Name	Description	Type
24	Reserved	–	–
25	Reserved	–	–
26	Reserved	–	–
27	Reserved	–	–
28	Reserved	–	–
29	Reserved	–	–
30	Ground	Ground	–
31	/C4	4.096 MHz clock	I/O
32	Ground	Ground	–
33	/F0	8 kHz framing signal	I/O
34	Ground	Ground	–
35	C2	2.048 MHz clock	I/O
36	Ground	Ground	–
37	SEC8K	Secondary 8 kHz timing	I/O
38	Ground	Ground	–
39	Reserved	–	–
40	Reserved	–	–

A.6 MVIP/C2 Termination Jumper

Connector J5 provides optional termination for the MVIP C2 clock signal.

Table A–6. MVIP/C2 Termination Jumper J5 Connector Pinout

J5 Pin No.	Signal Name	Description	Type
1	MVIP_C2O	MVIP C2 clock	O
2	TERM_C2O	Termination (1000pf/ 1K)	–
3	NC	–	–

A.7 MVIP/C4 Termination Jumper

Connector J6 provides optional termination for the MVIP C4 clock signal.

Table A–7. MVIP/C4 Termination Jumper J6 Connector Pinout

J6 Pin No.	Signal Name	Description	Type
1	MVIP_C4O	MVIP /C4 clock	O
2	TERM_C4B	Termination (1000pf/ 1K)	–
3	NC	–	–

A.8 CPLD ISP Header

Connector J7 provides the CPLD's JTAG in-system programming port that allows the McEVM's onboard logic to be reprogrammed. This connector is a 10-pin header (two rows of five pins) with connections shown in Table A–8 to communicate with the Altera ByteBlaster™ parallel port cable. The 10-pin female connector on the cable is connected to the male header on the McEVM. The pins have 0.025-inch square posts with 0.100-inch spacing.

The J7 CPLD ISP connector is for factory use and is not installed.

Table A–8. CPLD ISP J7 Pinout

J7 Pin No.	Signal Name	Description	Type
1	TCK	Test clock	I
2	GND	Ground	–
3	TDO	Test data output	O
4	VCC	5 V	O
5	TMS	Test mode select	I
6	NC	–	–
7	NC	–	–
8	NC	–	–
9	TDI	Test data input	I
10	GND	–	–

A.9 Expansion Memory Interface Connector

Connector J8 provides the ’C6201 asynchronous expansion memory interface signals to a daughterboard that can provide additional memory and memory-mapped devices.

Table A–9. Expansion Memory Interface J8 Connector Pinout

J8 Pin No.	Signal Name	Type	J6 Pin No.	Signal Name	Type
1	5 V	O	2	5 V	O
3	XA21	O	4	XA20	O
5	XA19	O	6	XA18	O
7	XA17	O	8	XA16	O
9	XA15	O	10	XA14	O
11	GND	–	12	GND	–
13	XA13	O	14	XA12	O
15	XA11	O	16	XA10	O
17	XA9	O	18	XA8	O
19	XA7	O	20	XA6	O
21	5 V	O	22	5 V	O
23	XA5	O	24	XA4	O
25	XA3	O	26	XA2	O
27	$\overline{\text{XBE3}}$	O	28	$\overline{\text{XBE2}}$	O
29	$\overline{\text{XBE1}}$	O	30	$\overline{\text{XBE0}}$	O
31	GND	–	32	GND	–
33	XD31	I/O/Z	34	XD30	I/O/Z
35	XD29	I/O/Z	36	XD28	I/O/Z
37	XD27	I/O/Z	38	XD26	I/O/Z
39	XD25	I/O/Z	40	XD24	I/O/Z
41	3.3 V	–	42	3.3 V	–
43	XD23	I/O/Z	44	XD22	I/O/Z
45	XD21	I/O/Z	46	XD20	I/O/Z
47	XD19	I/O/Z	48	XD18	I/O/Z
49	XD17	I/O/Z	50	XD16	I/O/Z
51	GND	–	52	GND	–
53	XD15	I/O/Z	54	XD14	I/O/Z
55	XD13	I/O/Z	56	XD12	I/O/Z
57	XD11	I/O/Z	58	XD10	I/O/Z
59	XD9	I/O/Z	60	XD8	I/O/Z
61	GND	–	62	GND	–
63	XD7	I/O/Z	64	XD6	I/O/Z
65	XD5	I/O/Z	66	XD4	I/O/Z
67	XD3	I/O/Z	68	XD2	I/O/Z
69	XD1	I/O/Z	70	XD0	I/O/Z
71	GND	–	72	GND	–
73	$\overline{\text{XRE}}$	O	74	$\overline{\text{XWE}}$	O
75	$\overline{\text{XOE}}$	O	76	XRDY	I
77	SPARE (N/C)	–	78	$\overline{\text{XCE1}}$	O
79	GND	–	80	GND	–

A.10 Expansion Peripheral Interface Connector

Connector J9 provides ’C6201 expansion peripheral interface signals to a daughterboard.

Table A–10. Expansion Peripheral Interface J9 Connector Pinout

J9 Pin No.	Signal Name	Type	J7 Pin No.	Signal Name	Type
1	12 V	O	2	–12 V	O
3	GND	–	4	GND	–
5	5 V	O	6	5 V	O
7	GND	–	8	GND	–
9	5 V	O	10	5 V	O
11	SPARE (N/C)	–	12	SPARE (N/C)	–
13	RSVD (N/C)	–	14	RSVD (N/C)	–
15	RSVD (N/C)	–	16	RSVD (N/C)	–
17	SPARE (N/C)	–	18	SPARE (N/C)	–
19	3.3 V	O	20	3.3 V	O
21	XCLKX0	I/O/Z	22	XCLKS0	I
23	XFSX0	I/O/Z	24	XDX0	O
25	GND	–	26	GND	–
27	XCLKR0	I/O/Z	28	SPARE (N/C)	–
29	XFSR0	I/O/Z	30	XDR0	I
31	GND	–	32	GND	–
33	XCLKX1	I/O/Z	34	XCLKS1	I
35	XFSX1	I/O/Z	36	XDX1	O
37	GND	–	38	GND	–
39	XCLKR1	I/O/Z	40	SPARE (N/C)	–
41	XFSR1	I/O/Z	42	XDR1	I
43	GND	–	44	GND	–
45	TOUT0	O	46	TINP0	O
47	SPARE (N/C)	–	48	SPARE (N/C)	–
49	TOUT1	I	50	TINP1	I
51	GND	–	52	GND	–
53	XEXT_INT7	I	54	IACK	O
55	INUM3	O	56	INUM2	O
57	INUM1	O	58	INUM0	O
59	$\overline{\text{XRESET}}$	O	60	DSP_PD	O
61	GND	–	62	GND	–
63	XCNTL1	O	64	XCNTL0	O
65	XSTAT1	I	66	XSTAT0	I
67	SPARE (N/C)	–	68	SPARE (N/C)	–
69	$\overline{\text{XCE2}}$	O	70	$\overline{\text{XCE3}}$	O
71	DMAC3	O	72	DMAC2	O
73	DMAC1	O	74	DMAC0	O
75	GND	–	76	GND	–
77	GND	–	78	XCLKOUT2	O
79	GND	–	80	GND	–

A.11 TMS320C62x JTAG Emulation Header

Connector J10 provides the ’C6201’s emulation port based on the IEEE 1149.1 standard. This connector is a 14-pin header (two rows of seven pins) with connections shown in Table A–11 to communicate with an XDS510 emulator. Pin 6 is used for keying to ensure a proper connection.

Table A–11. TMS320C62x JTAG Emulation Header J10 Pinout

J10 Pin No.	Signal Name	Description	Type
1	TMS	Test mode select	I
2	$\overline{\text{TRST}}$	Test reset	I
3	TDI	Test data input	I
4	GND	Ground	–
5	PD (V_{CC})	Presence detect. Indicates that the emulation cable is connected and the target is powered up. PD is tied to 3.3 V on the McEVM.	O
6	KEY	Not used. This pin is cut off on the J5 header. This pin is filled in on the XDS510 connector.	–
7	TDO	Test data out	O
8	GND	Ground	–
9	TCK_RET	Test clock return. Test clock input to the emulator.	O
10	GND	Ground	–
11	TCK	Test clock. TCK is a 10.368-MHz clock source from the emulation cable pod.	I
12	GND	Ground	–
13	EMU0	Emulation pin 0	I/O
14	EMU1	Emulation pin 1	I/O

A.12 External Power Connector

Connector J11 enables the ’C62x McEVM to be connected to an external power supply during stand-alone operation.

Table A–12. External Power J11 Connector Pinout

J11 Pin No.	Signal Name	Description	Type
1	12	12 V_{DC} at 500 mA	I
2	–12	–12 V_{DC} at 100 mA	I
3	GND	Ground	–
4	5	5 V_{DC} at 4 A	I

A.13 DSP Fan Power Connector

Connector J12 provides power to the DSP cooling fan (not used). This 2-pin connector provides 5 V at 100 mA to the fan.

Table A–13. DSP Fan Power J12 Connector Pinout

J12 Pin No.	Signal Name	Description	Type
1	GND	Ground	–
2	PWR	5 V_{DC} at 100 mA	O

A.14 PCI Local Bus Connector

Connector P1 provides the PCI local bus to the ’C62x McEVM.

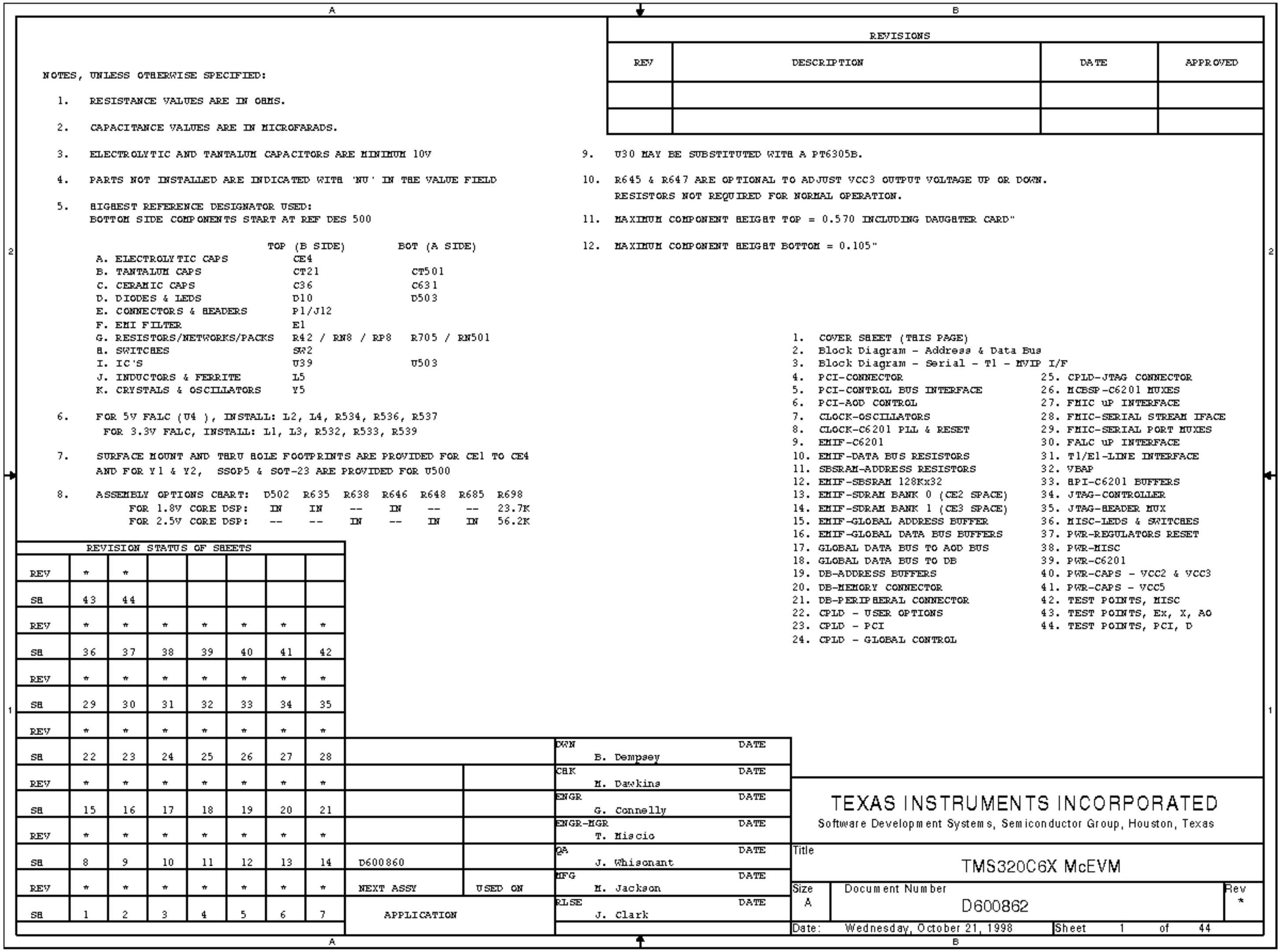
Table A–14. PCI Local Bus P1 Connector Pinout

P1 Pin No.	Side B	Side A	P1 Pin No.	Side B	Side A
1	–12 V	TRST#	32	AD17	AD16
2	TCK	12 V	33	C/BE2#	3.3 V
3	GND	TMS	34	GND	FRAME#
4	TDO	TDI	35	IRDY#	GND
5	5 V	5 V	36	3.3 V	TRDY#
6	5 V	RSVD	37	DEVSEL#	GND
7	INTB#	INTC#	38	GND	STOP#
8	INTD#	INTA#	39	LOCK#	3.3 V
9	PRSNT1#	RSVD	40	PERR#	SDONE
10	RSVD	5 V	41	3.3 V	SBO#
11	PRSNT2#	RSVD	42	SERR#	GND
12	GND	GND	43	3.3 V	PAR
13	GND	GND	44	C/BE1#	AD15
14	RSVD	RSVD	45	AD14	3.3 V
15	GND	RST#	46	GND	AD13
16	CLK	5 V	47	AD12	AD11
17	GND	GNT#	48	AD10	GND
18	REQ#	GND	49	GND	AD9
19	5 V	RSVD	50	Key	Key
20	AD31	AD30	51	Key	Key
21	AD29	3.3 V	52	AD8	C/BE0#
22	GND	AD28	53	AD7	3.3 V
23	AD27	AD26	54	3.3 V	AD6
24	AD25	GND	55	AD5	AD4
25	3.3 V	AD24	56	AD3	GND
26	C/BE3#	TDSEL	57	GND	AD2
27	AD23	3.3 V	58	AD1	AD0
28	GND	AD20	59	5 V	5 V
29	AD21	GND	60	ACK64#	REQ64#
30	AD19	AD18	61	5 V	5 V
31	3.3 V	AD16	62	5 V	5 V

Appendix B

TMS320C6x McEVM Schematics

This appendix contains the schematics for the TMS320C6x McEVM.



REVISIONS			
REV	DESCRIPTION	DATE	APPROVED

NOTES, UNLESS OTHERWISE SPECIFIED:

1. RESISTANCE VALUES ARE IN OHMS.

2. CAPACITANCE VALUES ARE IN MICROFARADS.

3. ELECTROLYTIC AND TANTALUM CAPACITORS ARE MINIMUM 10V

4. PARTS NOT INSTALLED ARE INDICATED WITH 'NU' IN THE VALUE FIELD

5. HIGHEST REFERENCE DESIGNATOR USED:
BOTTOM SIDE COMPONENTS START AT REF DES 500

	TOP (B SIDE)	BOT (A SIDE)
A. ELECTROLYTIC CAPS	CE4	
B. TANTALUM CAPS	CT21	CT501
C. CERAMIC CAPS	C36	C631
D. DIODES & LEDS	D10	D503
E. CONNECTORS & HEADERS	P1/J12	
F. EMI FILTER	E1	
G. RESISTORS/NETWORKS/PACKS	R42 / RN8 / RP8	R705 / RN501
H. SWITCHES	SW2	
I. IC'S	U39	U503
J. INDUCTORS & FERRITE	L5	
K. CRYSTALS & OSCILLATORS	Y5	

6. FOR 5V FALC (U4), INSTALL: L2, L4, R534, R536, R537
FOR 3.3V FALC, INSTALL: L1, L3, R532, R533, R539

7. SURFACE MOUNT AND THRU HOLE FOOTPRINTS ARE PROVIDED FOR CE1 TO CE4
AND FOR Y1 & Y2, SSOP5 & SOT-23 ARE PROVIDED FOR U500

8. ASSEMBLY OPTIONS CHART:

	D502	R635	R638	R646	R648	R685	R698
FOR 1.8V CORE DSP:	IN	IN	--	IN	--	--	23.7K
FOR 2.5V CORE DSP:	--	--	IN	--	IN	IN	56.2K

9. U30 MAY BE SUBSTITUTED WITH A PT6305B.

10. R645 & R647 ARE OPTIONAL TO ADJUST VCC3 OUTPUT VOLTAGE UP OR DOWN.
RESISTORS NOT REQUIRED FOR NORMAL OPERATION.

11. MAXIMUM COMPONENT HEIGHT TOP = 0.570 INCLUDING DAUGHTER CARD"

12. MAXIMUM COMPONENT HEIGHT BOTTOM = 0.105"

REVISION STATUS OF SHEETS							
REV	*	*					
SH	43	44					
REV	*	*	*	*	*	*	*
SH	36	37	38	39	40	41	42
REV	*	*	*	*	*	*	*
SH	29	30	31	32	33	34	35
REV	*	*	*	*	*	*	*
SH	22	23	24	25	26	27	28
REV	*	*	*	*	*	*	*
SH	15	16	17	18	19	20	21
REV	*	*	*	*	*	*	*
SH	8	9	10	11	12	13	14
REV	*	*	*	*	*	*	*
SH	1	2	3	4	5	6	7

D600860	
NEXT ASSY	USED ON
APPLICATION	

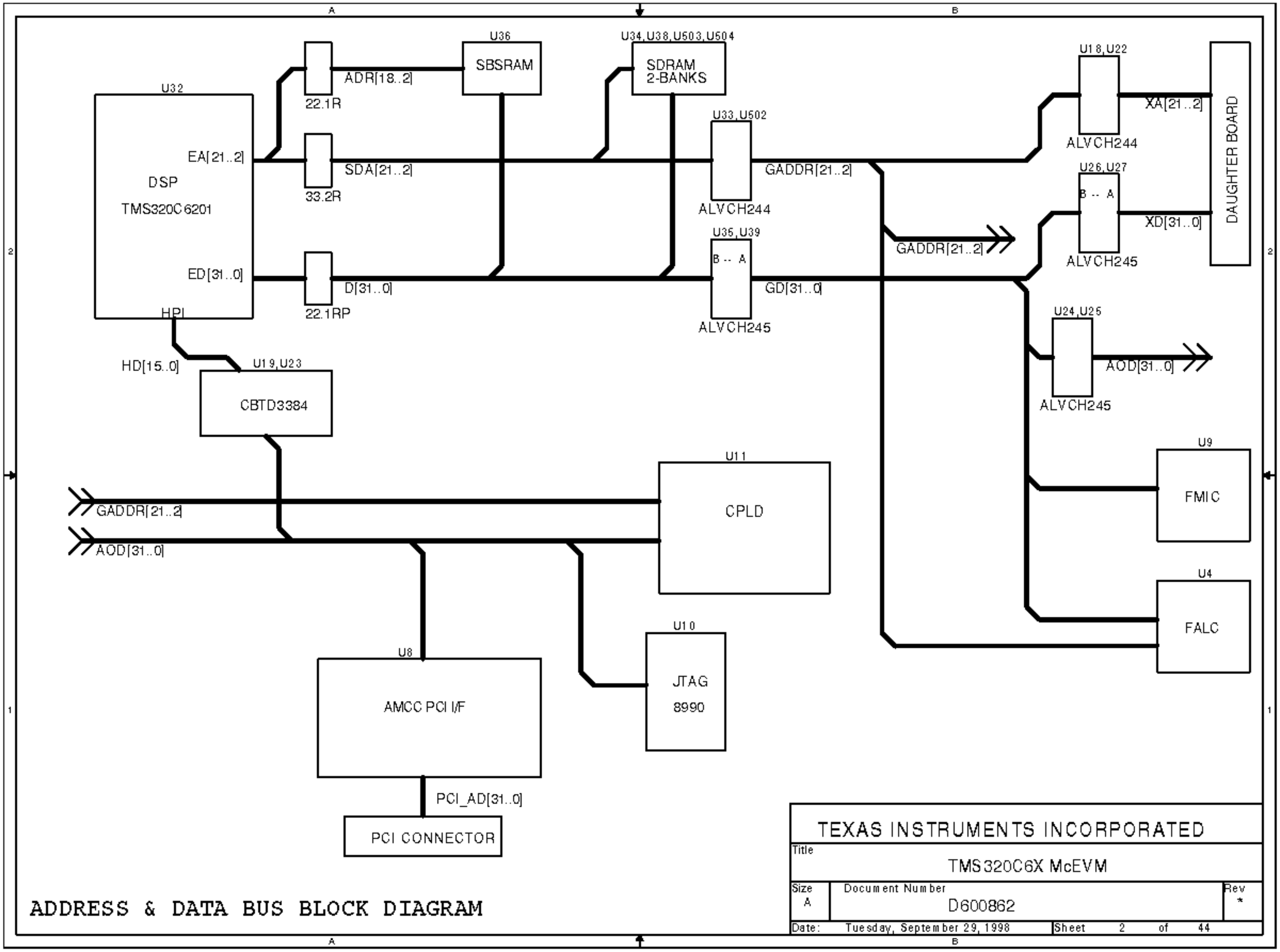
DWN	B. Dempsey	DATE
CHK	M. Dawkins	DATE
ENGR	G. Connelly	DATE
ENGR-MGR	T. Miscio	DATE
QA	J. Whisonant	DATE
MFG	M. Jackson	DATE
RLSE	J. Clark	DATE

TEXAS INSTRUMENTS INCORPORATED
Software Development Systems, Semiconductor Group, Houston, Texas

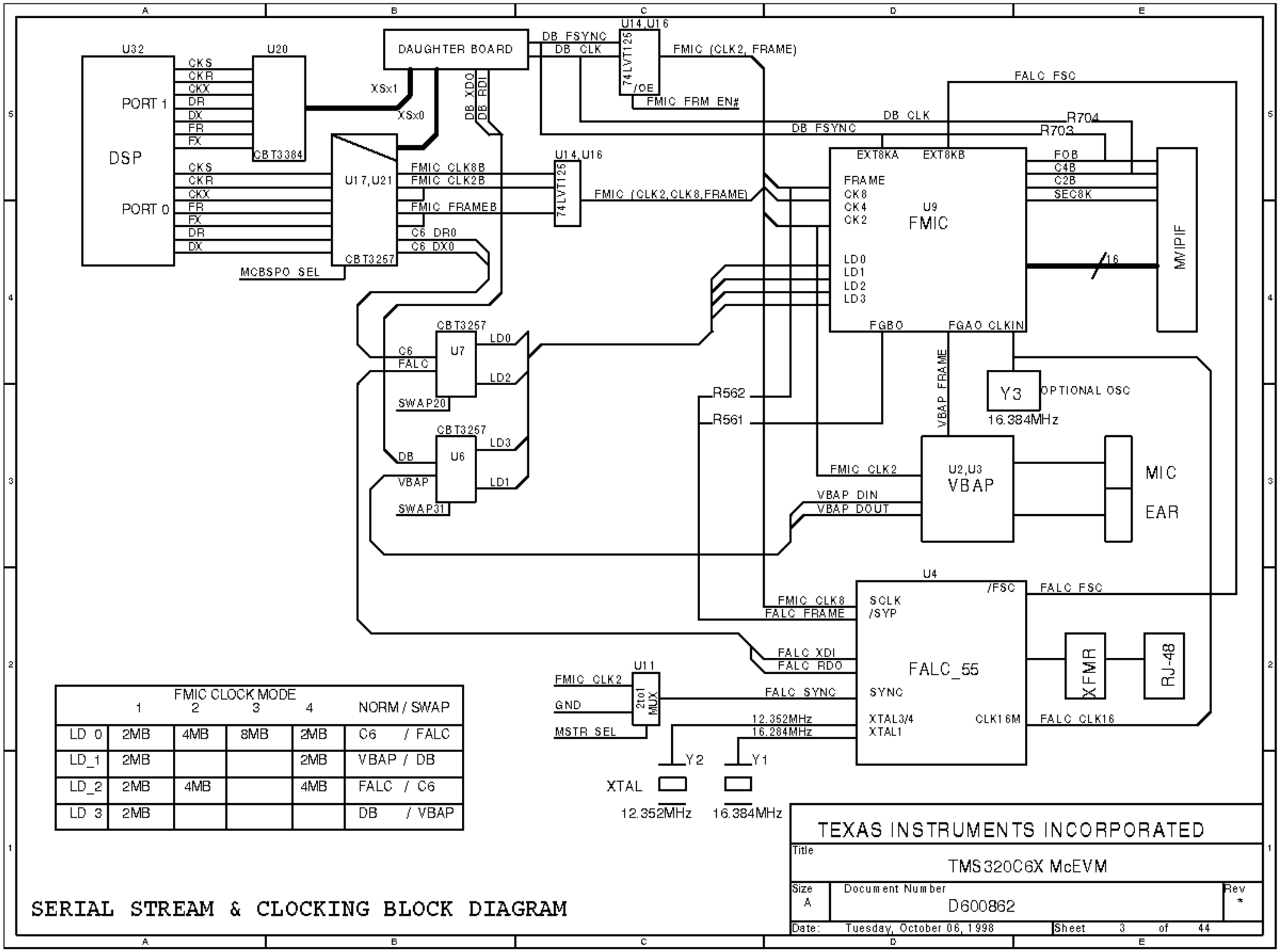
Title
TMS320C6X McEVM

Size	Document Number	Rev
A	D600862	*

Date: Wednesday, October 21, 1998 | Sheet 1 of 44



A
B
2
1
U32
DSP
TMS320C6201
EA[21..2]
ED[31..0]
HPI
22.1R
ADR[18..2]
U36
SBSRAM
U34,U38,U503,U504
SDRAM
2-BANKS
33.2R
SDA[21..2]
22.1RP
D[31..0]
U33,U502
ALVCH244
GADDR[21..2]
U35,U39
B -- A
ALVCH245
GD[31..0]
U18,U22
ALVCH244
XA[21..2]
U26,U27
B -- A
ALVCH245
XD[31..0]
DAUGHTER BOARD
GADDR[21..2]
U24,U25
ALVCH245
AOD[31..0]
HD[15..0]
U19,U23
CBTD3384
U11
CPLD
GADDR[21..2]
AOD[31..0]
U9
FMIC
U4
FALC
U10
JTAG
8990
U8
AMCC PCI I/F
PCI_AD[31..0]
PCI CONNECTOR
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Tuesday, September 29, 1998
Sheet 2 of 44
ADDRESS & DATA BUS BLOCK DIAGRAM



FMIC CLOCK MODE					
	1	2	3	4	NORM / SWAP
LD 0	2MB	4MB	8MB	2MB	C6 / FALC
LD_1	2MB			2MB	VBAP / DB
LD_2	2MB	4MB		4MB	FALC / C6
LD 3	2MB				DB / VBAP

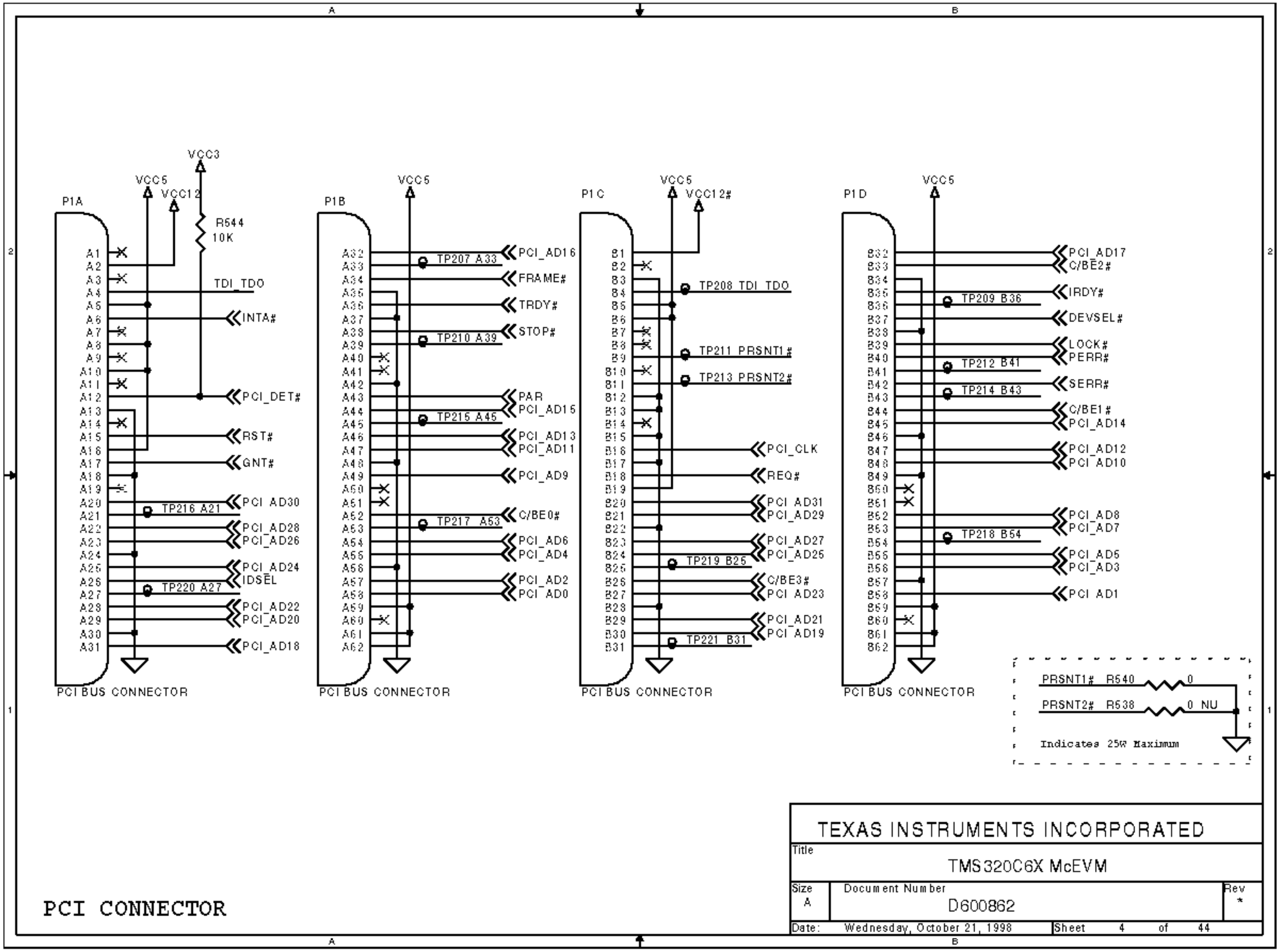
SERIAL STREAM & CLOCKING BLOCK DIAGRAM

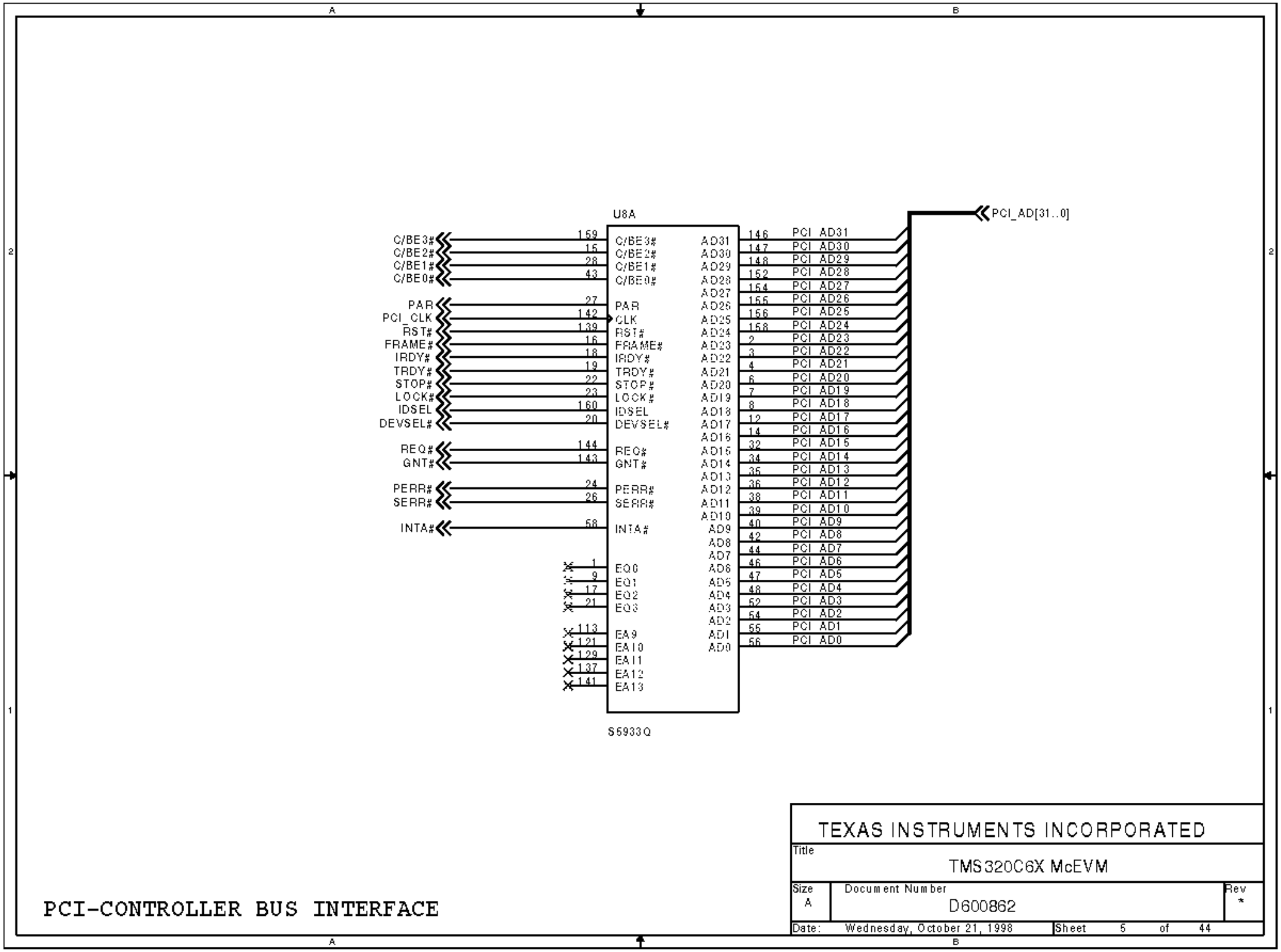
TEXAS INSTRUMENTS INCORPORATED

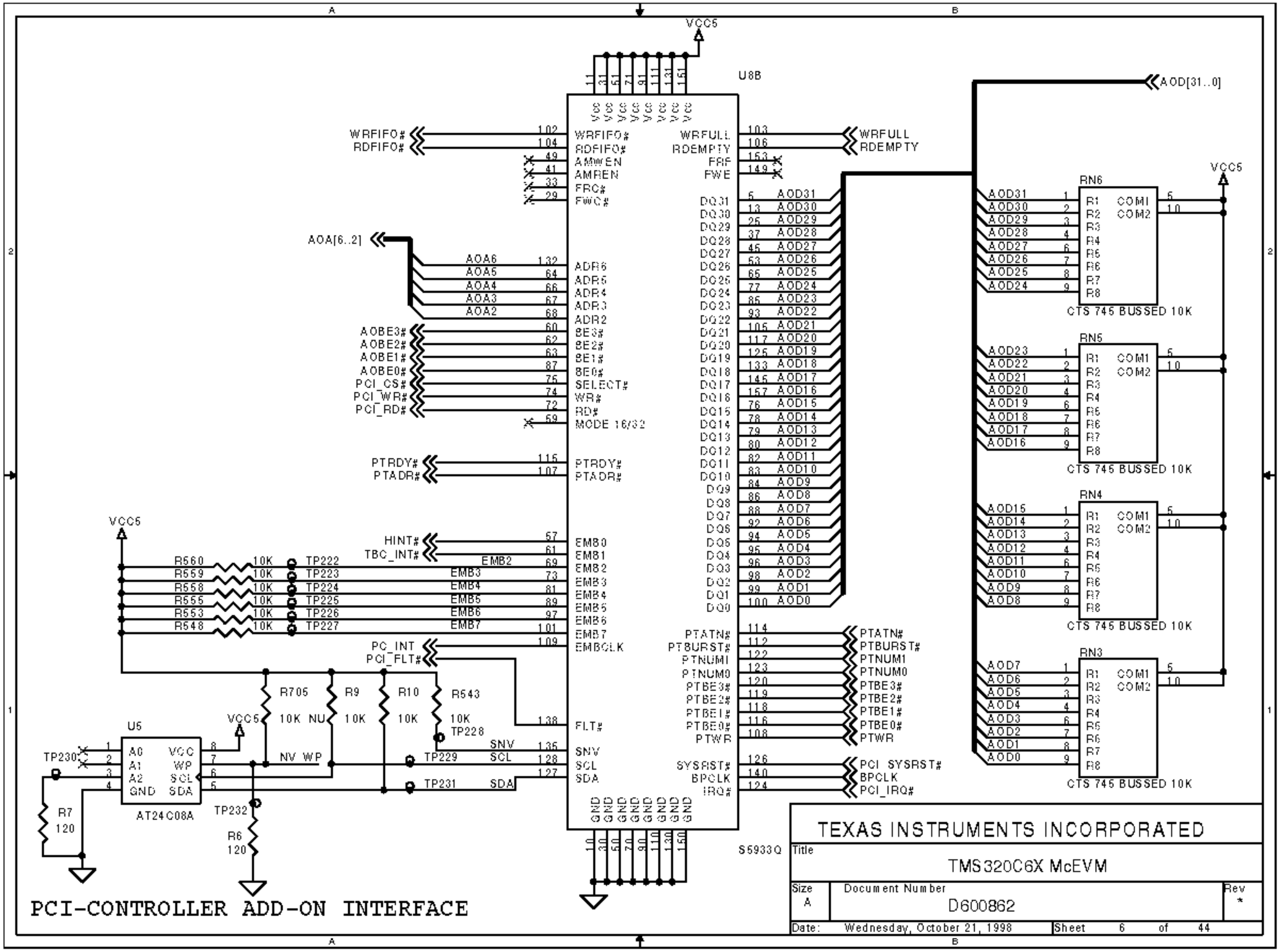
Title
TMS320C6X McEVM

Size A | Document Number D600862 | Rev *

Date: Tuesday, October 06, 1998 | Sheet 3 of 44

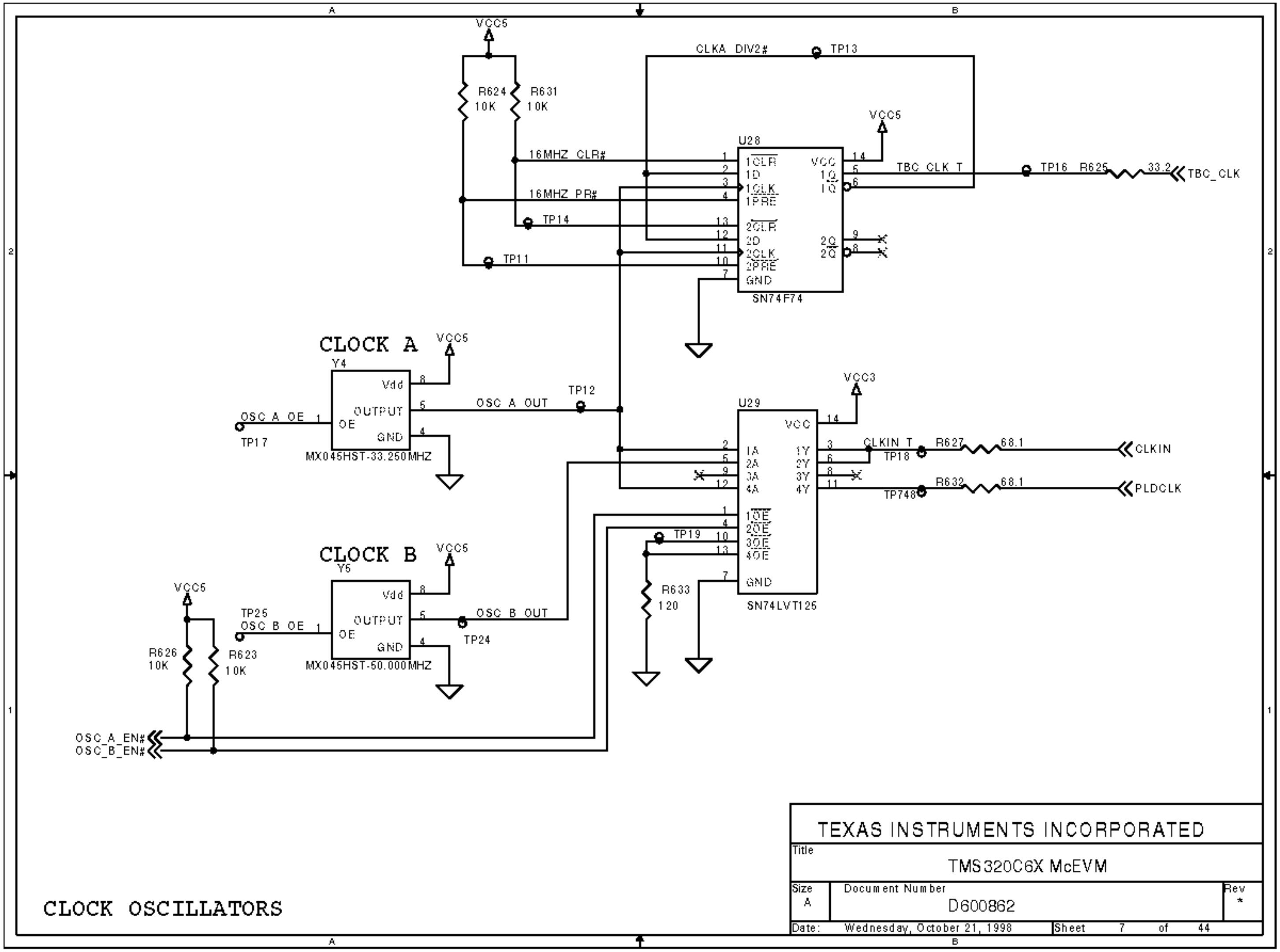

PCI CONNECTOR
P1A
P1B
P1C
P1D
PCI BUS CONNECTOR
VCC3
VCC5
VCC12
VCC12#
R544
10K
TDI_TDO
INTA#
PCI_DET#
RST#
GNT#
TP216 A21
TP220 A27
PCI_AD30
PCI_AD28
PCI_AD26
PCI_AD24
IDSEL
PCI_AD22
PCI_AD20
PCI_AD18
TP207 A33
TP210 A39
TP215 A45
TP217 A53
PCI_AD16
FRAME#
TRDY#
STOP#
PAR
PCI_AD15
PCI_AD13
PCI_AD11
PCI_AD9
C/BE0#
PCI_AD6
PCI_AD4
PCI_AD2
PCI_AD0
TP208 TDI_TDO
TP211 PRSNT1#
TP213 PRSNT2#
TP219 B25
TP221 B31
PCI_CLK
REQ#
PCI_AD31
PCI_AD29
PCI_AD27
PCI_AD25
C/BE3#
PCI_AD23
PCI_AD21
PCI_AD19
TP209 B36
TP212 B41
TP214 B43
TP218 B54
PCI_AD17
C/BE2#
IRDY#
DEVSEL#
LOCK#
PERR#
SERR#
C/BE1#
PCI_AD14
PCI_AD12
PCI_AD10
PCI_AD8
PCI_AD7
PCI_AD5
PCI_AD3
PCI_AD1
PRSNT1# R540 0
PRSNT2# R538 0 NU
Indicates 25W Maximum
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 4 of 44

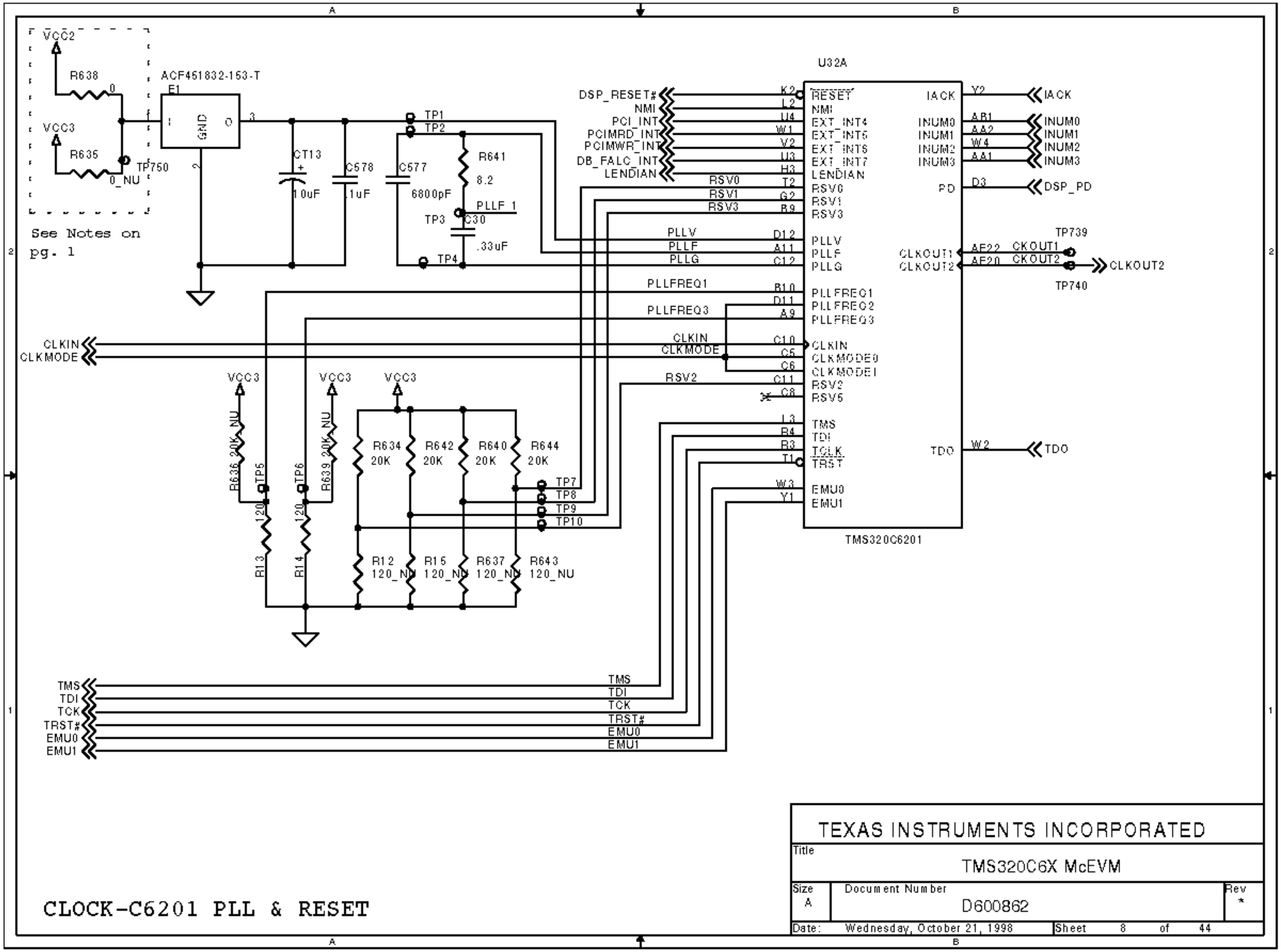

A
B
2
1
U8A
S5933Q
PCI_AD[31..0]
C/BE3# 159 C/BE3#
C/BE2# 15 C/BE2#
C/BE1# 28 C/BE1#
C/BE0# 43 C/BE0#
PAR 27 PAR
PCI_CLK 142 CLK
RST# 139 RST#
FRAME# 16 FRAME#
IRDY# 18 IRDY#
TRDY# 19 TRDY#
STOP# 22 STOP#
LOCK# 23 LOCK#
IDSEL 160 IDSEL
DEVSEL# 20 DEVSEL#
REQ# 144 REQ#
GNT# 143 GNT#
PERR# 24 PERR#
SERR# 26 SERR#
INTA# 58 INTA#
1 EQ0
9 EQ1
17 EQ2
21 EQ3
113 EA9
121 EA10
129 EA11
137 EA12
141 EA13
AD31 146 PCI_AD31
AD30 147 PCI_AD30
AD29 148 PCI_AD29
AD28 152 PCI_AD28
AD27 154 PCI_AD27
AD26 155 PCI_AD26
AD25 156 PCI_AD25
AD24 158 PCI_AD24
AD23 2 PCI_AD23
AD22 3 PCI_AD22
AD21 4 PCI_AD21
AD20 6 PCI_AD20
AD19 7 PCI_AD19
AD18 8 PCI_AD18
AD17 12 PCI_AD17
AD16 14 PCI_AD16
AD15 32 PCI_AD15
AD14 34 PCI_AD14
AD13 35 PCI_AD13
AD12 36 PCI_AD12
AD11 38 PCI_AD11
AD10 39 PCI_AD10
AD9 40 PCI_AD9
AD8 42 PCI_AD8
AD7 44 PCI_AD7
AD6 46 PCI_AD6
AD5 47 PCI_AD5
AD4 48 PCI_AD4
AD3 52 PCI_AD3
AD2 54 PCI_AD2
AD1 55 PCI_AD1
AD0 56 PCI_AD0
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 5 of 44
PCI-CONTROLLER BUS INTERFACE

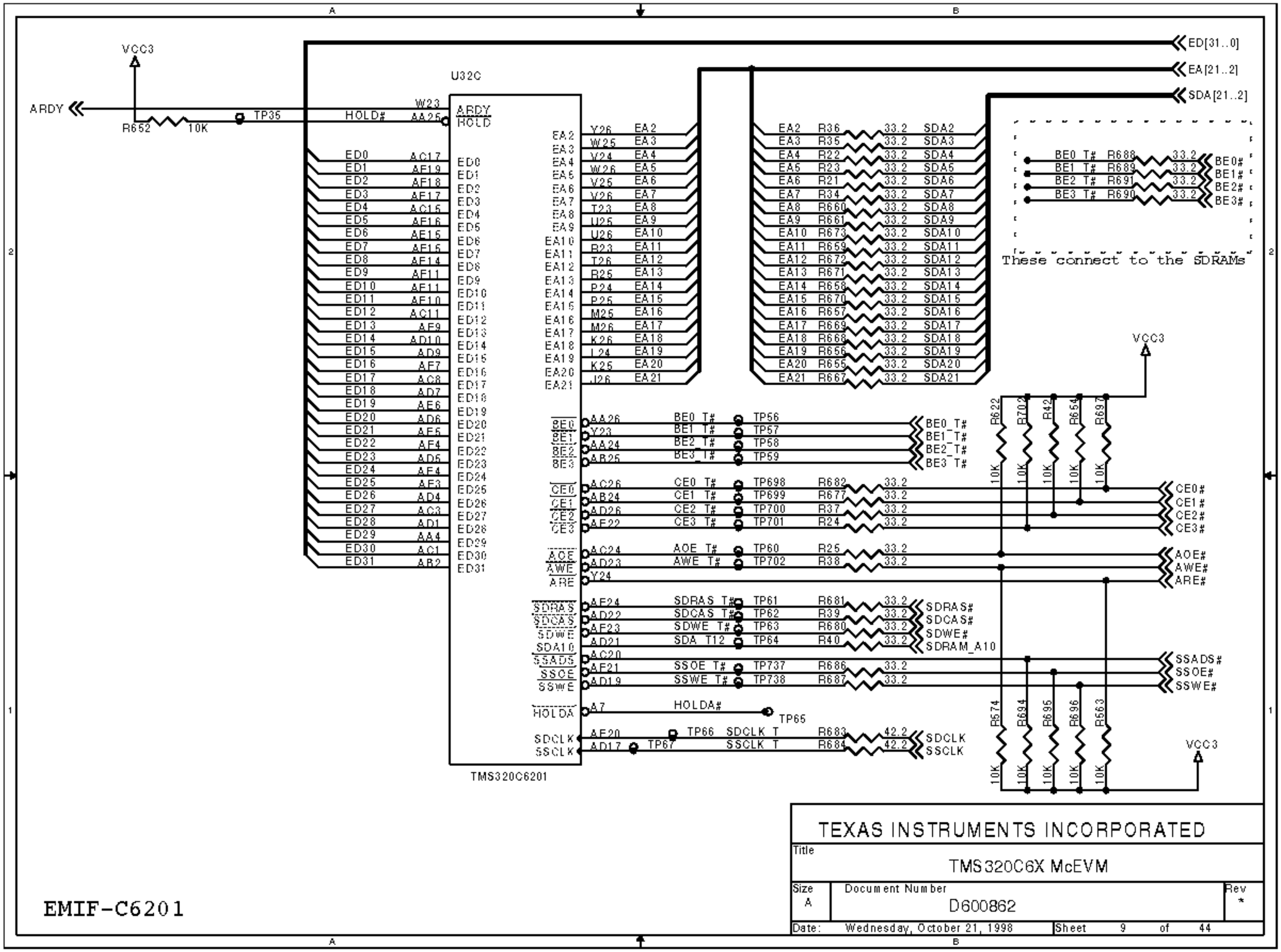


TEXAS INSTRUMENTS INCORPORATED

Title	TMS320C6X McEVM		
Size A	Document Number D600862		Rev *
Date: Wednesday, October 21, 1998	Sheet 6 of 44		


CLOCK A
CLOCK B
CLOCK OSCILLATORS
VCC5
VCC3
R624 10K
R631 10K
R626 10K
R623 10K
R633 120
R625 33.2
R627 68.1
R632 68.1
16MHZ_CLR#
16MHZ_PR#
CLKA_DIV2#
TP11
TP12
TP13
TP14
TP16
TP17
TP18
TP19
TP24
TP25
TP748
U28
1CLR
1D
1CLK
1PRE
2CLR
2D
2CLK
2PRE
GND
VCC
1Q
2Q
SN74F74
TBC_CLK_T
TBC_CLK
Y4
Vdd
OUTPUT
OE
GND
OSC_A_OE
OSC_A_OUT
MX045HST-33.250MHZ
Y5
OSC_B_OE
OSC_B_OUT
MX045HST-50.000MHZ
U29
1A
2A
3A
4A
1Y
2Y
3Y
4Y
1OE
2OE
3OE
4OE
SN74LVT125
CLKIN_T
CLKIN
PLDCLK
OSC_A_EN#
OSC_B_EN#
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 7 of 44


VCC2
R638
0
ACF451832-153-T
E1
GND
3
2
VCC3
R635
0_NU
TP750
CT13
10uF
C578
.1uF
C577
6800pF
TP1
TP2
R641
8.2
PLLF_1
TP3
C30
.33uF
TP4
See Notes on pg. 1
DSP_RESET#
NMI
PCI_INT
PCIMRD_INT
PCIMWR_INT
DB_FALC_INT
LENDIAN
RSV0
RSV1
RSV3
PLLV
PLLF
PLLG
PLLFREQ1
PLLFREQ3
CLKIN
CLKMODE
RSV2
U32A
K2 RESET
L2 NMI
U4 EXT_INT4
W1 EXT_INT5
V2 EXT_INT6
U3 EXT_INT7
H3 LENDIAN
T2 RSV0
G2 RSV1
B9 RSV3
D12 PLLV
A11 PLLF
C12 PLLG
B10 PLLFREQ1
D11 PLLFREQ2
A9 PLLFREQ3
C10 CLKIN
C5 CLKMODE0
C6 CLKMODE1
C11 RSV2
C8 RSV5
L3 TMS
B4 TDI
B3 TCLK
T1 TRST
W3 EMU0
Y1 EMU1
IACK Y2
INUM0 AB1
INUM1 AA2
INUM2 W4
INUM3 AA1
PD D3
CLKOUT1 AF22
CLKOUT2 AF20
TDO W2
TMS320C6201
IACK
INUM0
INUM1
INUM2
INUM3
DSP_PD
TP739
CKOUT1
CKOUT2
CLKOUT2
TP740
TDO
VCC3
VCC3
VCC3
R636 20K_NU
R639 20K_NU
TP5
TP6
120
120
R13
R14
R634 20K
R642 20K
R640 20K
R644 20K
TP7
TP8
TP9
TP10
R12 120_NU
R15 120_NU
R637 120_NU
R643 120_NU
TMS
TDI
TCK
TRST#
EMU0
EMU1
A
B
1
2
CLOCK-C6201 PLL & RESET
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 8 of 44



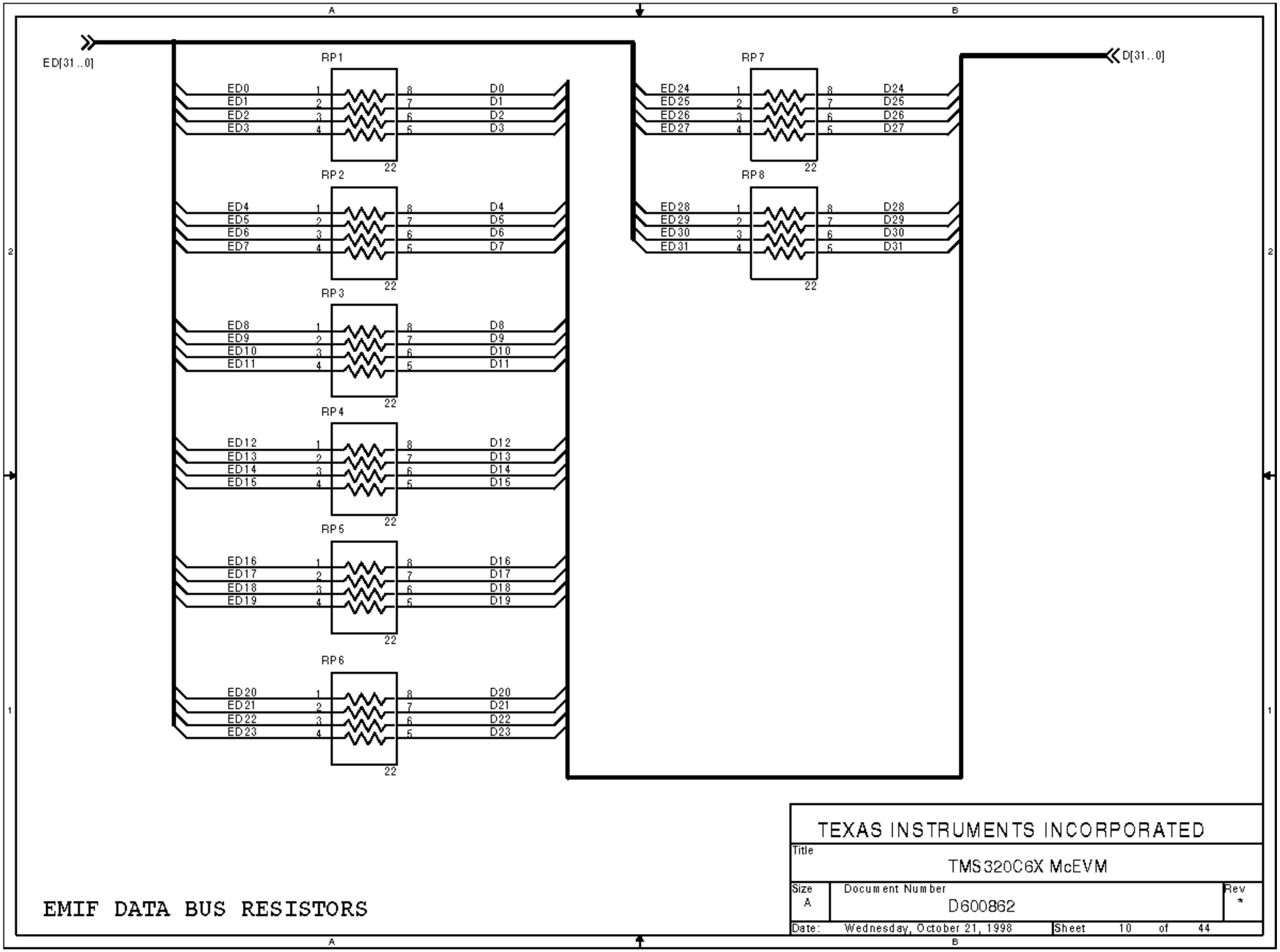
TEXAS INSTRUMENTS INCORPORATED

Title
TMS320C6X McEVM

Size	Document Number	Rev
A	D600862	*

Date: Wednesday, October 21, 1998 Sheet 9 of 44

EMIF-C6201



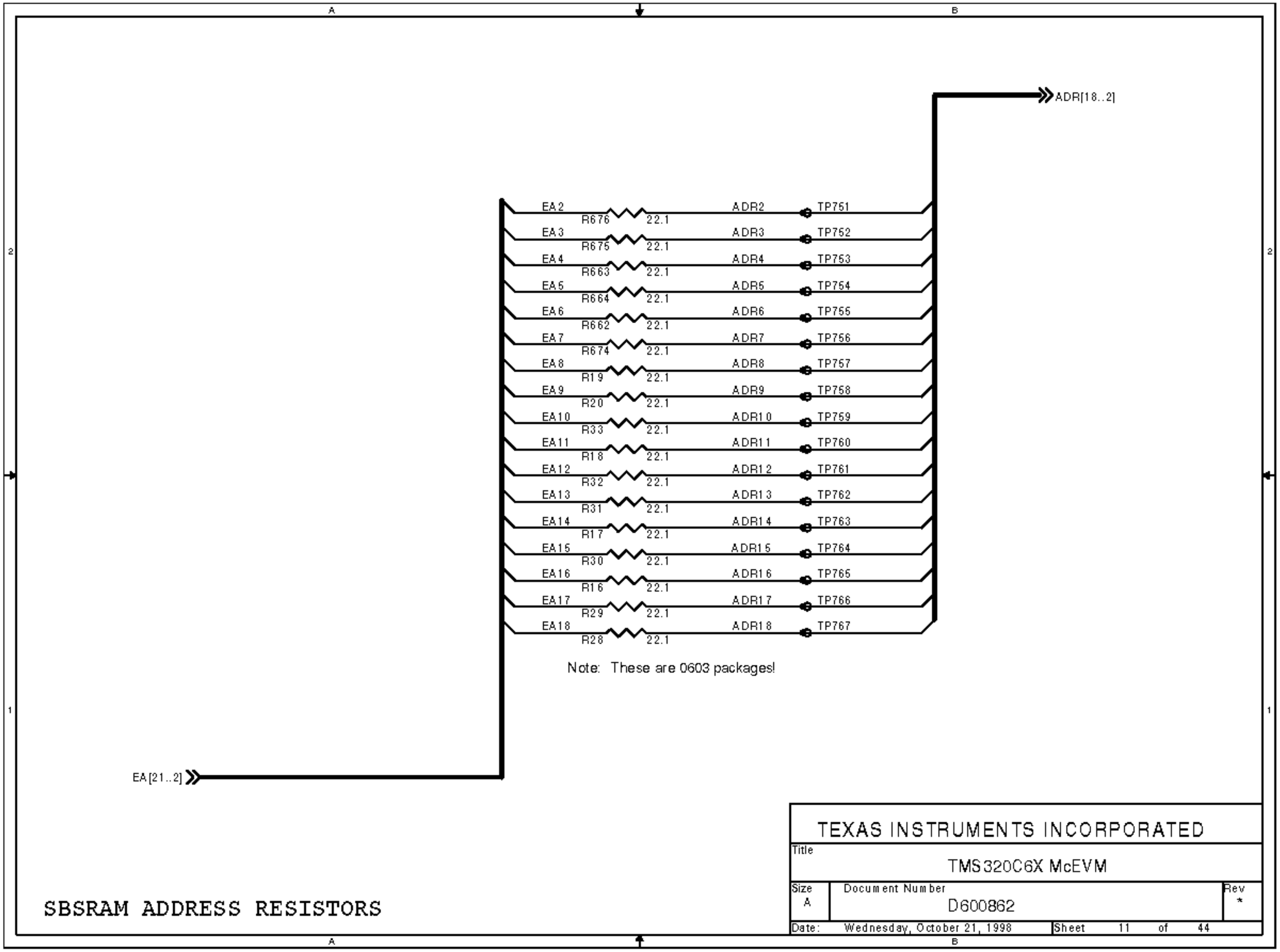
EMIF DATA BUS RESISTORS

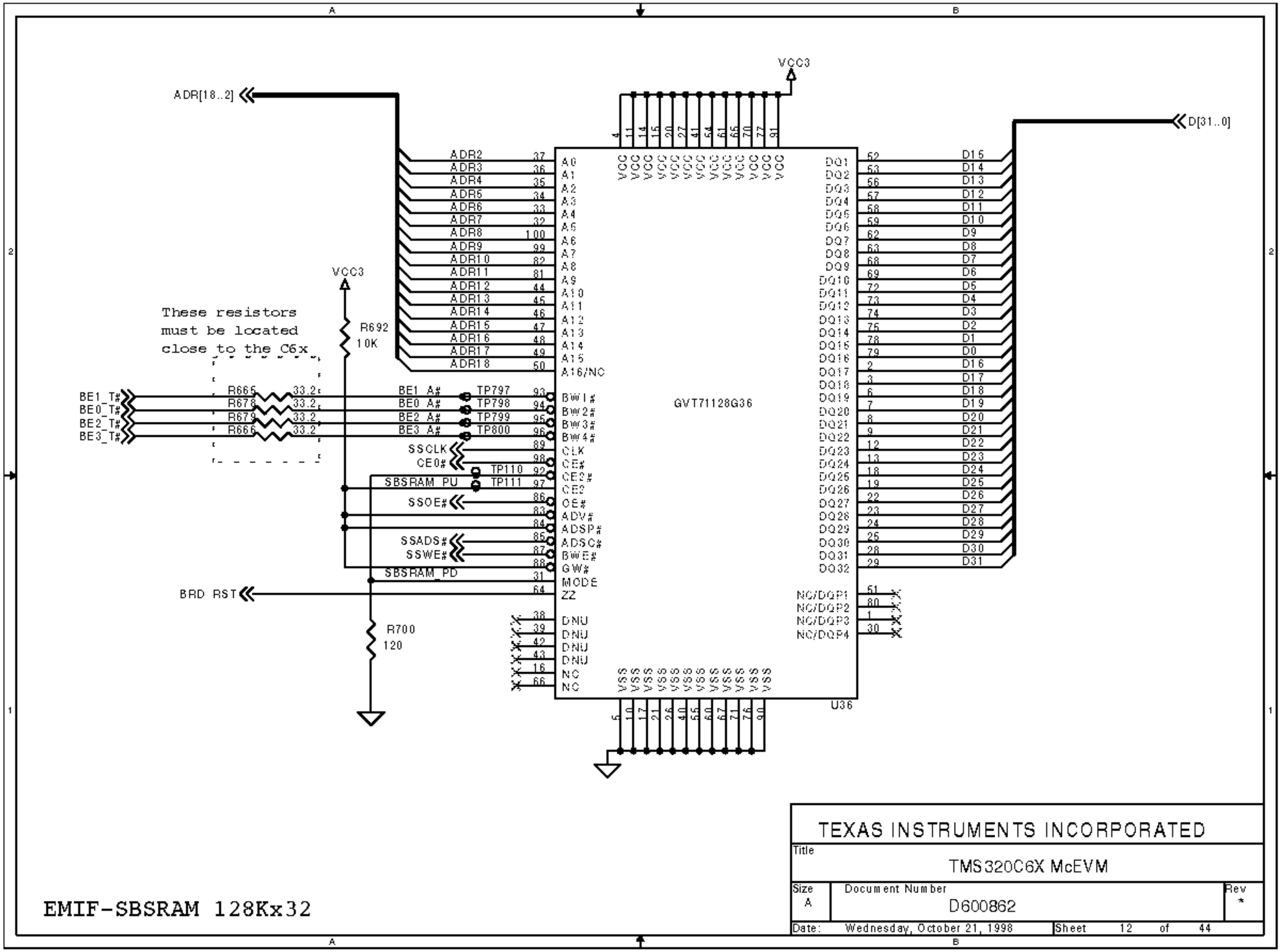
TEXAS INSTRUMENTS INCORPORATED

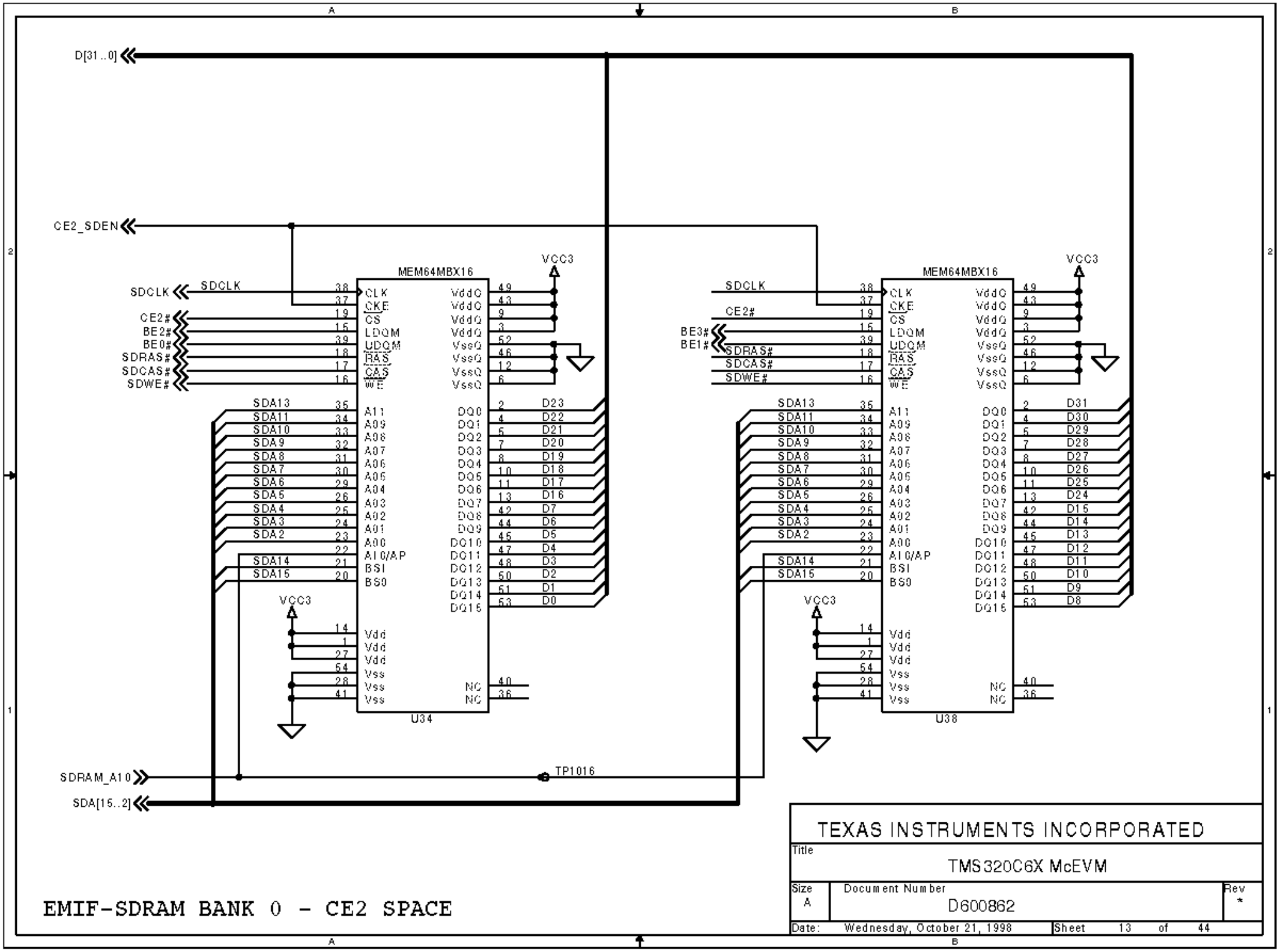
Title TMS320C6X McEVM

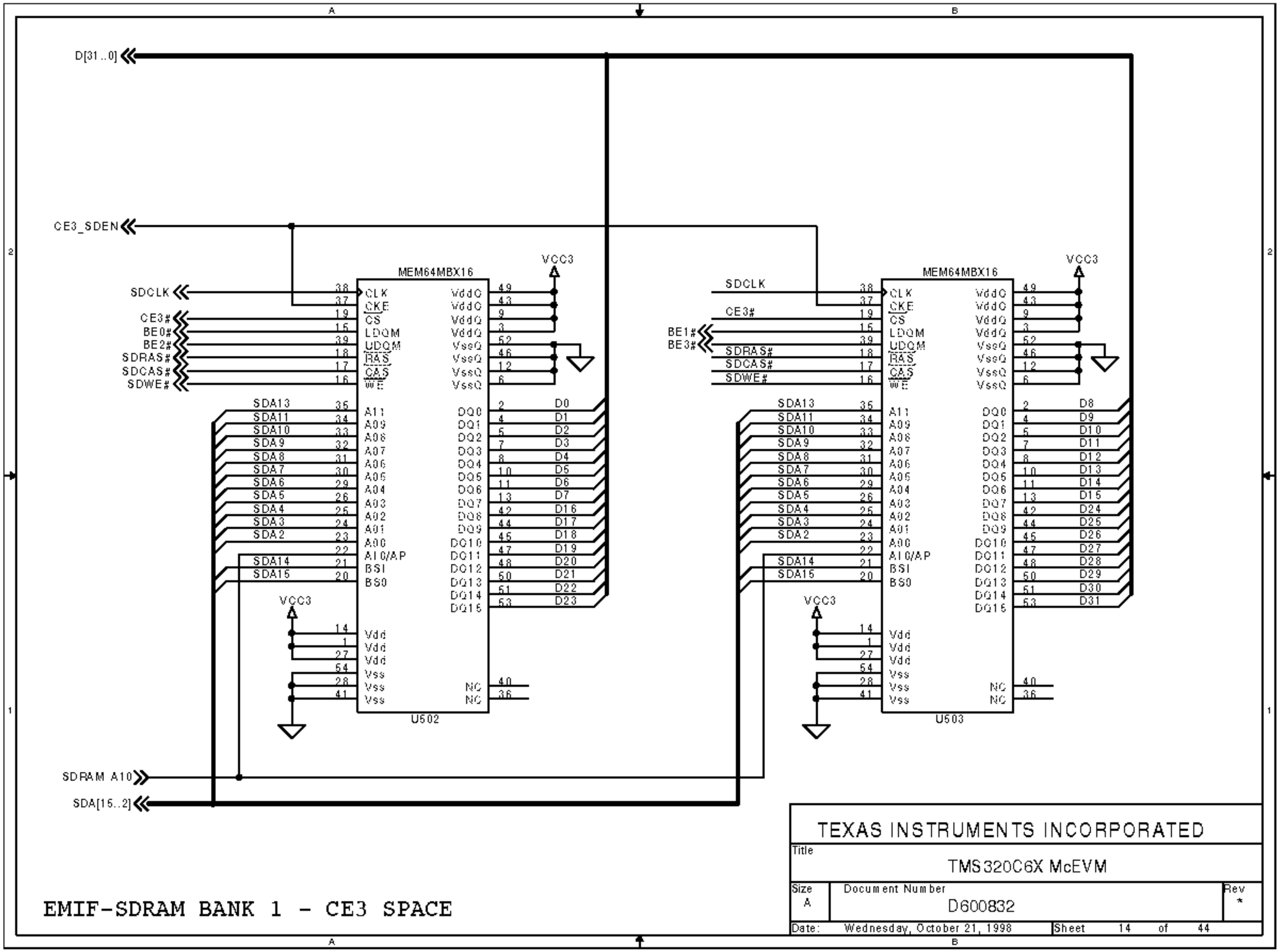
Size A Document Number D600862 Rev *

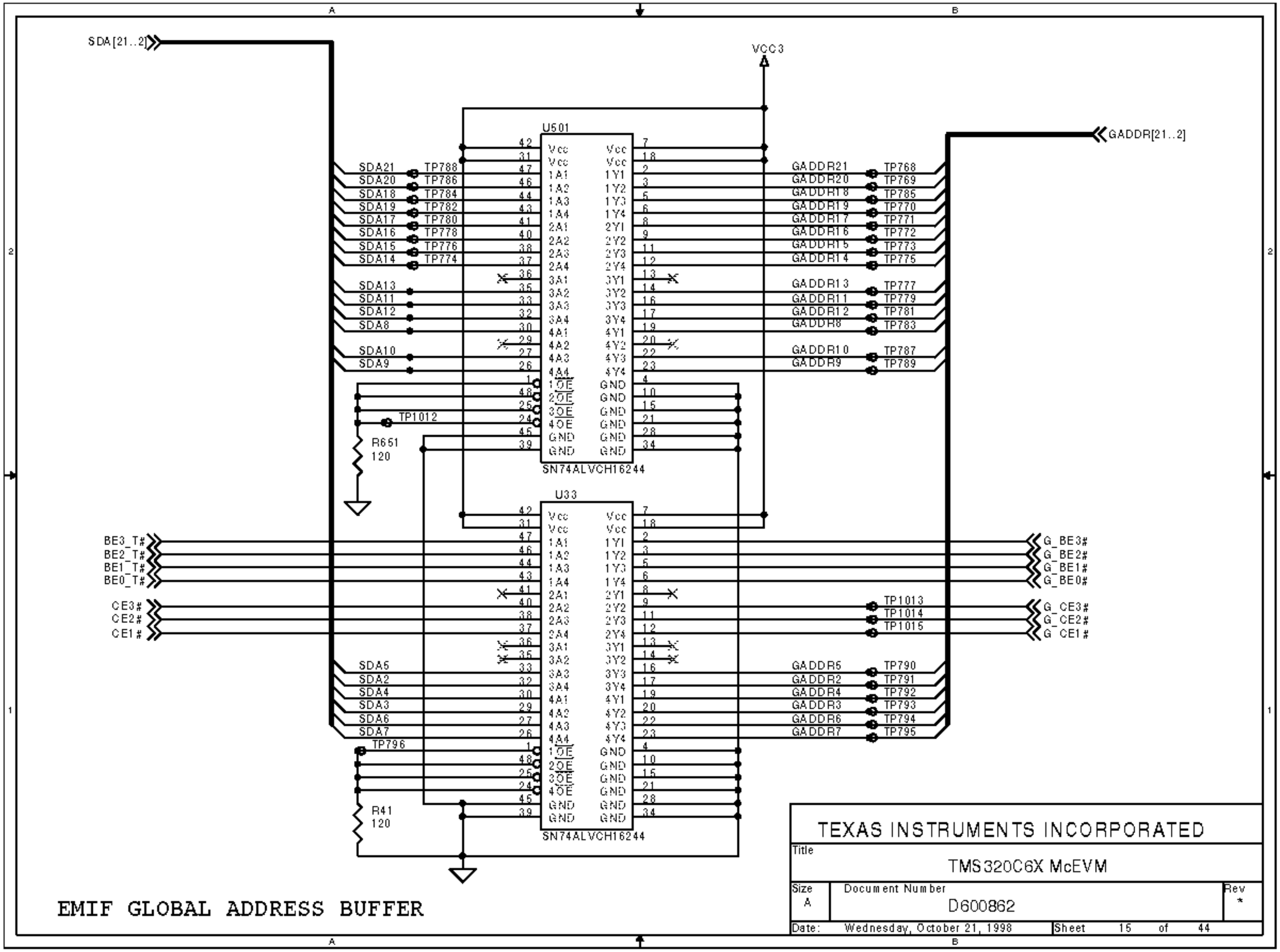
Date: Wednesday, October 21, 1998 Sheet 10 of 44

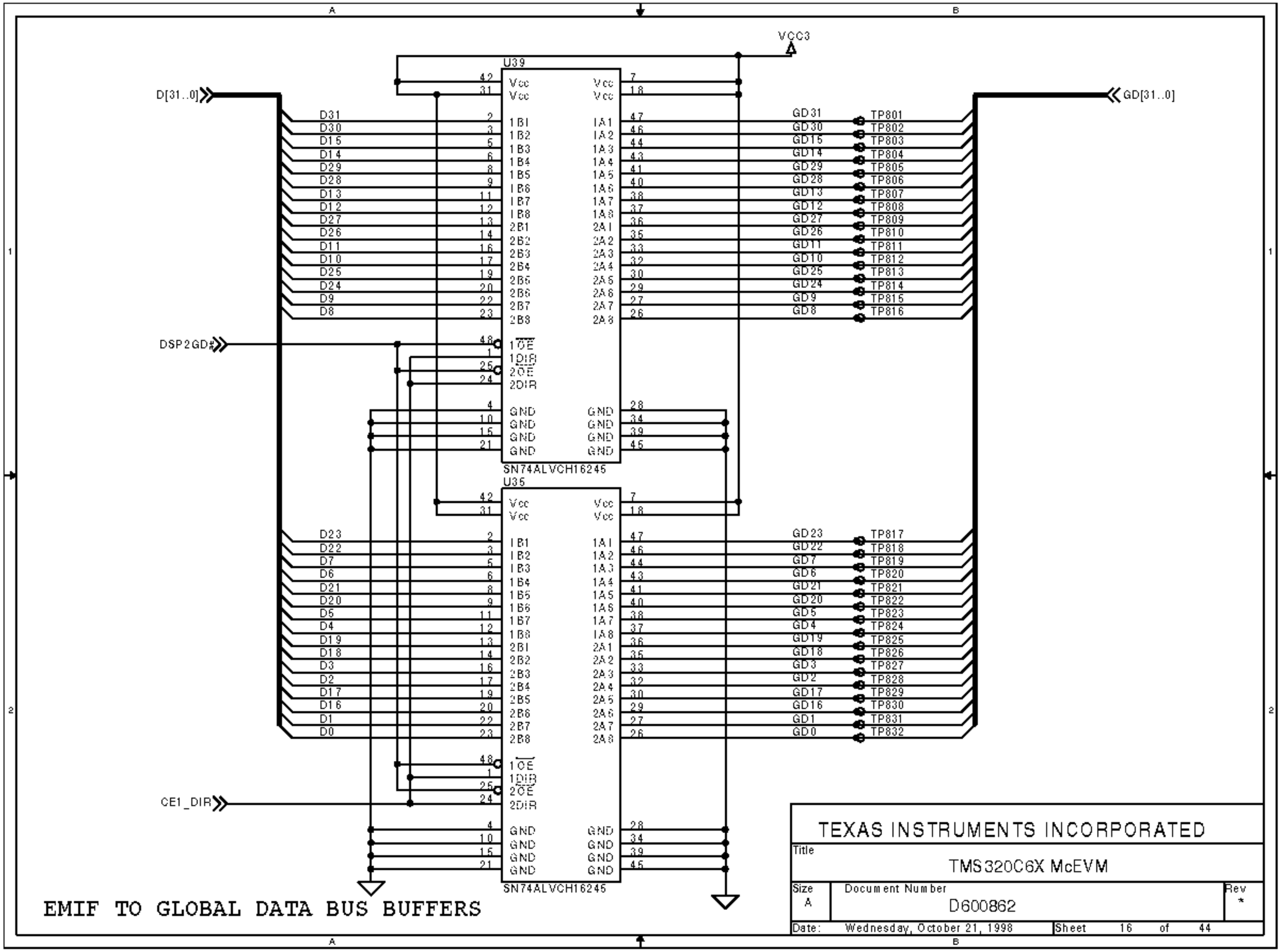

A
B
2
1
ADR[18..2]
EA2 R676 22.1 ADR2 TP751
EA3 R675 22.1 ADR3 TP752
EA4 R663 22.1 ADR4 TP753
EA5 R664 22.1 ADR5 TP754
EA6 R662 22.1 ADR6 TP755
EA7 R674 22.1 ADR7 TP756
EA8 R19 22.1 ADR8 TP757
EA9 R20 22.1 ADR9 TP758
EA10 R33 22.1 ADR10 TP759
EA11 R18 22.1 ADR11 TP760
EA12 R32 22.1 ADR12 TP761
EA13 R31 22.1 ADR13 TP762
EA14 R17 22.1 ADR14 TP763
EA15 R30 22.1 ADR15 TP764
EA16 R16 22.1 ADR16 TP765
EA17 R29 22.1 ADR17 TP766
EA18 R28 22.1 ADR18 TP767
Note: These are 0603 packages!
EA[21..2]
SBSRAM ADDRESS RESISTORS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 11 of 44

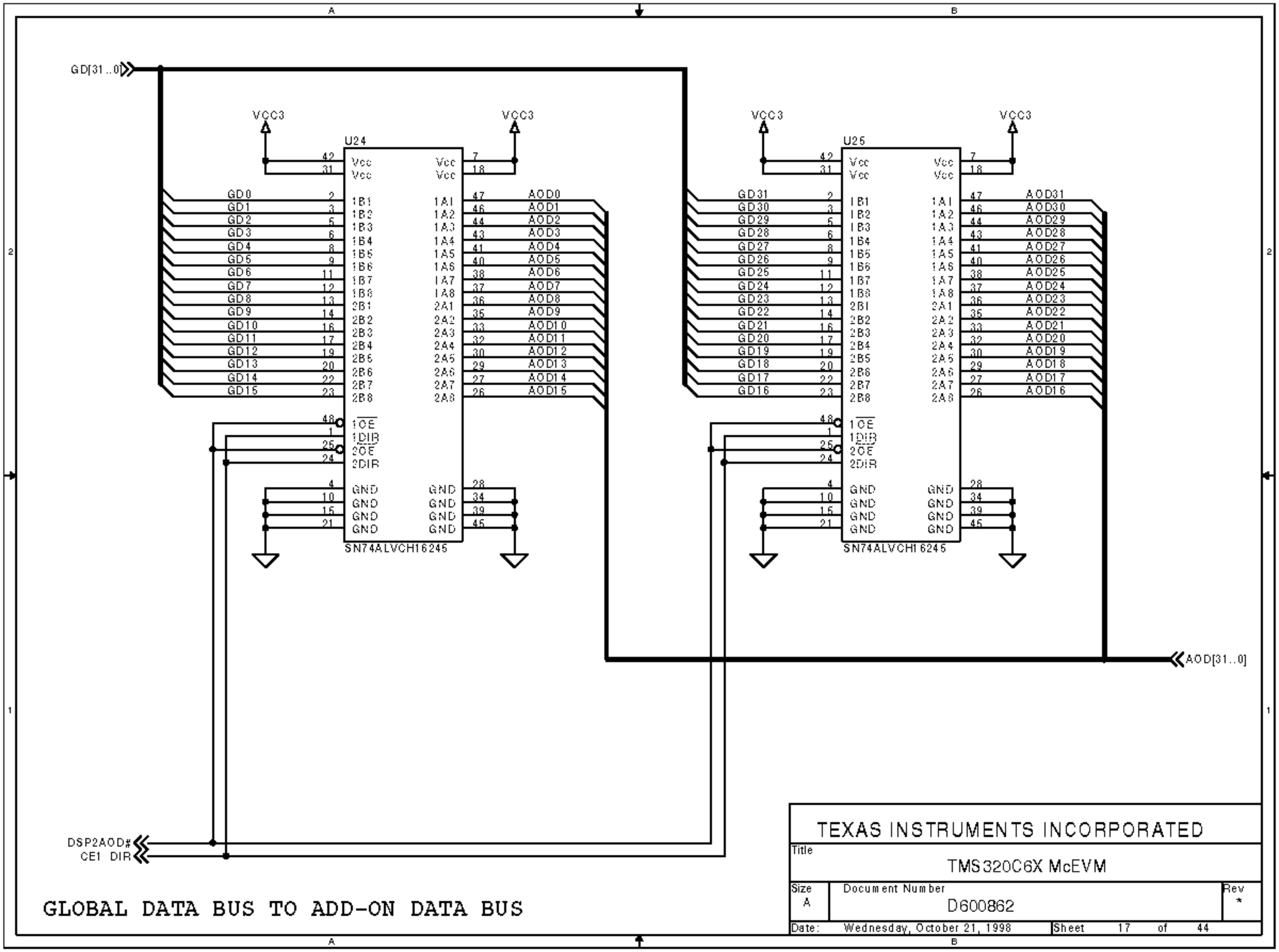

A
B
2
1
ADR[18..2]
D[31..0]
VCC3
4 11 14 15 20 27 41 54 61 65 70 77 91
VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC
ADR2 37 A0
ADR3 36 A1
ADR4 35 A2
ADR5 34 A3
ADR6 33 A4
ADR7 32 A5
ADR8 100 A6
ADR9 99 A7
ADR10 82 A8
ADR11 81 A9
ADR12 44 A10
ADR13 45 A11
ADR14 46 A12
ADR15 47 A13
ADR16 48 A14
ADR17 49 A15
ADR18 50 A16/NC
DQ1 52 D15
DQ2 53 D14
DQ3 56 D13
DQ4 57 D12
DQ5 58 D11
DQ6 59 D10
DQ7 62 D9
DQ8 63 D8
DQ9 68 D7
DQ10 69 D6
DQ11 72 D5
DQ12 73 D4
DQ13 74 D3
DQ14 75 D2
DQ15 78 D1
DQ16 79 D0
DQ17 2 D16
DQ18 3 D17
DQ19 6 D18
DQ20 7 D19
DQ21 8 D20
DQ22 9 D21
DQ23 12 D22
DQ24 13 D23
DQ25 18 D24
DQ26 19 D25
DQ27 22 D26
DQ28 23 D27
DQ29 24 D28
DQ30 25 D29
DQ31 28 D30
DQ32 29 D31
NC/DQP1 51
NC/DQP2 80
NC/DQP3 1
NC/DQP4 30
GVT71128G36
U36
These resistors must be located close to the C6x
VCC3
R692
10K
BE1_T# R665 33.2 BE1_A# TP797 93 BW1#
BE0_T# R678 33.2 BE0_A# TP798 94 BW2#
BE2_T# R679 33.2 BE2_A# TP799 95 BW3#
BE3_T# R666 33.2 BE3_A# TP800 96 BW4#
SSCLK 89 CLK
CE0# 98 CE#
TP110 92 CE2#
SBSRAM_PU TP111 97 CE2
SSOE# 86 OE#
83 ADV#
84 ADSP#
SSADS# 85 ADSC#
SSWE# 87 BWE#
88 GW#
SBSRAM_PD 31 MODE
BRD_RST 64 ZZ
38 DNU
39 DNU
42 DNU
43 DNU
16 NC
66 NC
R700
120
VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS
5 10 17 21 26 40 55 60 67 71 76 90
EMIF-SBSRAM 128Kx32
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 12 of 44


D[31..0]
CE2_SDEN
SDCLK
CE2#
BE2#
BE0#
SDRAS#
SDCAS#
SDWE#
BE3#
BE1#
MEM64MBX16
VCC3
CLK
CKE
CS
LDQM
UDQM
RAS
CAS
WE
VddQ
VssQ
A11
A09
A08
A07
A06
A05
A04
A03
A02
A01
A00
A10/AP
BS1
BS0
Vdd
Vss
NC
DQ0
DQ15
SDA13
SDA11
SDA10
SDA9
SDA8
SDA7
SDA6
SDA5
SDA4
SDA3
SDA2
SDA14
SDA15
D23
D22
D21
D20
D19
D18
D17
D16
D7
D6
D5
D4
D3
D2
D1
D0
D31
D30
D29
D28
D27
D26
D25
D24
D15
D14
D13
D12
D11
D10
D9
D8
U34
U38
TP1016
SDRAM_A10
SDA[15..2]
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 13 of 44
EMIF-SDRAM BANK 0 - CE2 SPACE


D[31..0]
CE3_SDEN
SDCLK
CE3#
BE0#
BE2#
SDRAS#
SDCAS#
SDWE#
BE1#
BE3#
MEM64MBX16
VCC3
CLK
CKE
CS
LDQM
UDQM
RAS
CAS
WE
VddQ
VssQ
SDA13
SDA11
SDA10
SDA9
SDA8
SDA7
SDA6
SDA5
SDA4
SDA3
SDA2
SDA14
SDA15
A11
A09
A08
A07
A06
A05
A04
A03
A02
A01
A00
A10/AP
BS1
BS0
DQ0
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQ8
DQ9
DQ10
DQ11
DQ12
DQ13
DQ14
DQ15
D0
D1
D2
D3
D4
D5
D6
D7
D16
D17
D18
D19
D20
D21
D22
D23
D8
D9
D10
D11
D12
D13
D14
D15
D24
D25
D26
D27
D28
D29
D30
D31
Vdd
Vss
NC
U502
U503
SDRAM A10
SDA[15..2]
EMIF-SDRAM BANK 1 - CE3 SPACE
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600832
Rev
*
Date: Wednesday, October 21, 1998
Sheet 14 of 44


SDA[21..2]
VCC3
GADDR[21..2]
U501
SN74ALVCH16244
U33
SN74ALVCH16244
Vcc
GND
1A1 1A2 1A3 1A4 2A1 2A2 2A3 2A4 3A1 3A2 3A3 3A4 4A1 4A2 4A3 4A4
1Y1 1Y2 1Y3 1Y4 2Y1 2Y2 2Y3 2Y4 3Y1 3Y2 3Y3 3Y4 4Y1 4Y2 4Y3 4Y4
1OE 2OE 3OE 4OE
SDA21 SDA20 SDA18 SDA19 SDA17 SDA16 SDA15 SDA14
SDA13 SDA11 SDA12 SDA8 SDA10 SDA9
SDA5 SDA2 SDA4 SDA3 SDA6 SDA7
TP788 TP786 TP784 TP782 TP780 TP778 TP776 TP774
GADDR21 GADDR20 GADDR18 GADDR19 GADDR17 GADDR16 GADDR15 GADDR14
GADDR13 GADDR11 GADDR12 GADDR8 GADDR10 GADDR9
GADDR5 GADDR2 GADDR4 GADDR3 GADDR6 GADDR7
TP768 TP769 TP785 TP770 TP771 TP772 TP773 TP775
TP777 TP779 TP781 TP783 TP787 TP789
TP790 TP791 TP792 TP793 TP794 TP795
TP1012
TP796
TP1013 TP1014 TP1015
R651
120
R41
120
BE3_T# BE2_T# BE1_T# BE0_T#
CE3# CE2# CE1#
G_BE3# G_BE2# G_BE1# G_BE0#
G_CE3# G_CE2# G_CE1#
A
B
1
2
EMIF GLOBAL ADDRESS BUFFER
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 15 of 44

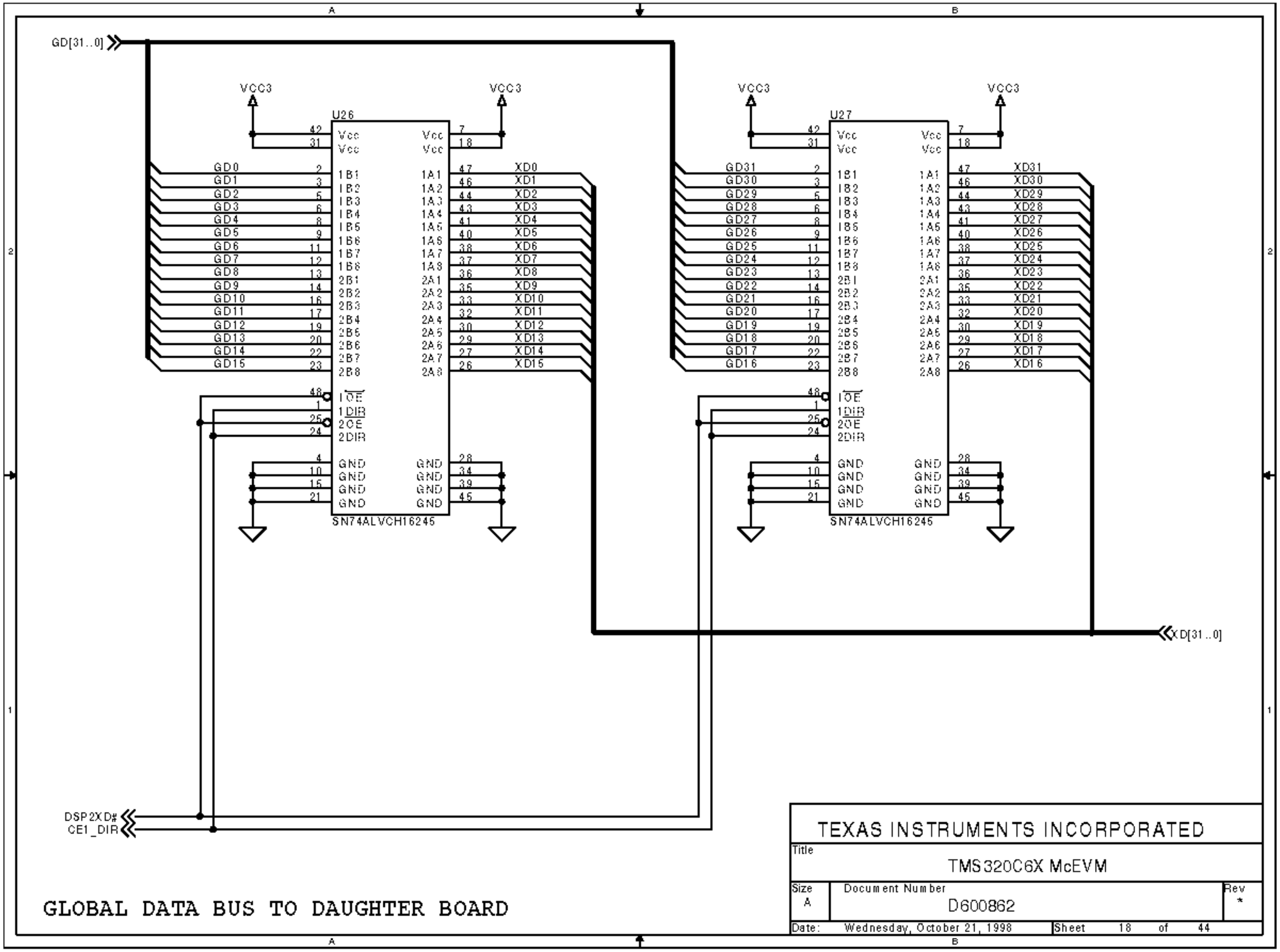

VCC3
U39
U35
SN74ALVCH16245
SN74ALVCH16245
D[31..0]
GD[31..0]
DSP2GD#
CE1_DIR
Vcc
GND
1OE
1DIR
2OE
2DIR
D31 D30 D15 D14 D29 D28 D13 D12 D27 D26 D11 D10 D25 D24 D9 D8
D23 D22 D7 D6 D21 D20 D5 D4 D19 D18 D3 D2 D17 D16 D1 D0
1B1 1B2 1B3 1B4 1B5 1B6 1B7 1B8 2B1 2B2 2B3 2B4 2B5 2B6 2B7 2B8
1A1 1A2 1A3 1A4 1A5 1A6 1A7 1A8 2A1 2A2 2A3 2A4 2A5 2A6 2A7 2A8
GD31 GD30 GD15 GD14 GD29 GD28 GD13 GD12 GD27 GD26 GD11 GD10 GD25 GD24 GD9 GD8
GD23 GD22 GD7 GD6 GD21 GD20 GD5 GD4 GD19 GD18 GD3 GD2 GD17 GD16 GD1 GD0
TP801 TP802 TP803 TP804 TP805 TP806 TP807 TP808 TP809 TP810 TP811 TP812 TP813 TP814 TP815 TP816
TP817 TP818 TP819 TP820 TP821 TP822 TP823 TP824 TP825 TP826 TP827 TP828 TP829 TP830 TP831 TP832
EMIF TO GLOBAL DATA BUS BUFFERS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 16 of 44
A
B
1
2



GLOBAL DATA BUS TO ADD-ON DATA BUS

TEXAS INSTRUMENTS INCORPORATED

Title	TMS320C6X McEVM	
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 17 of 44	



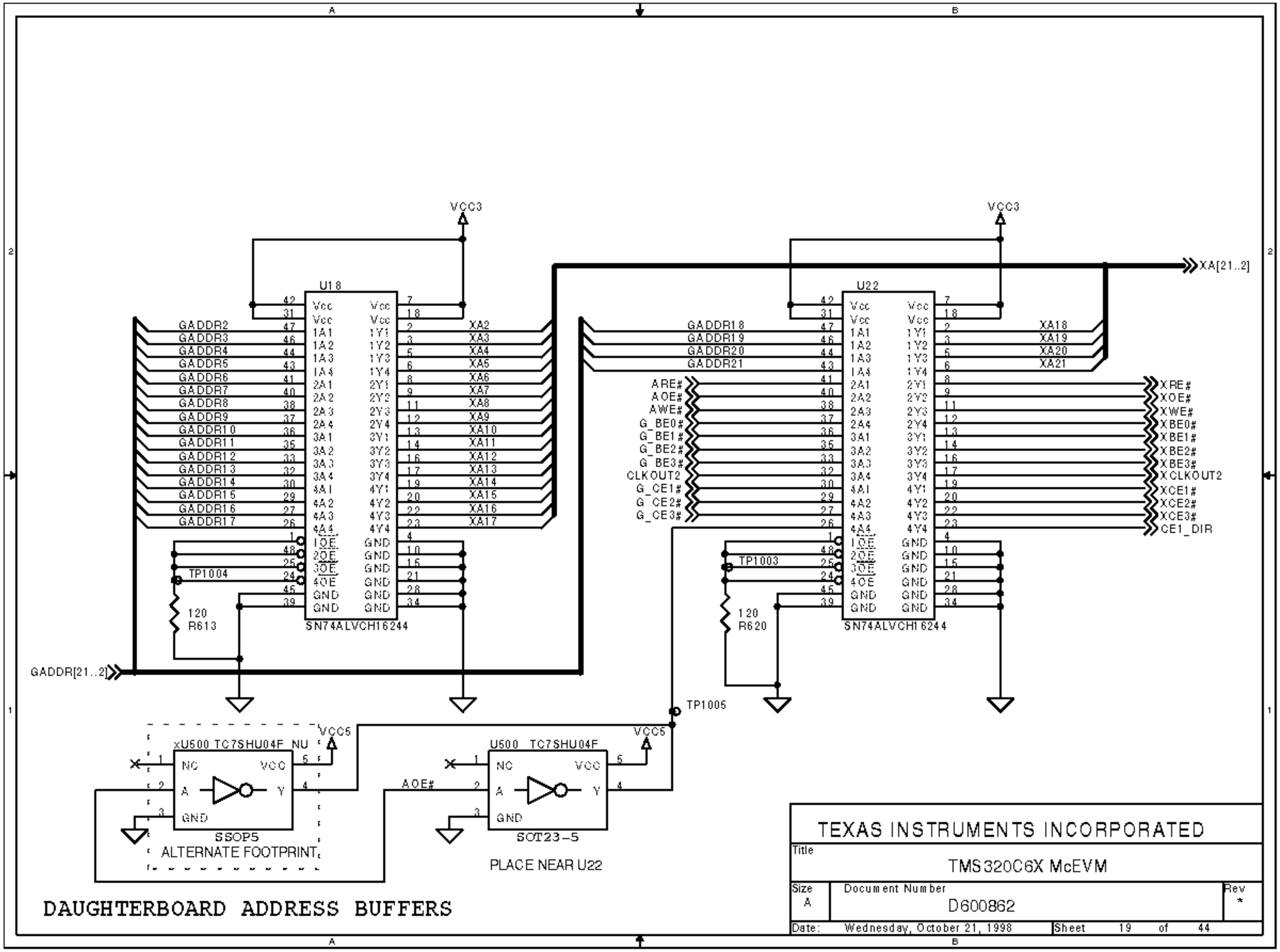
GLOBAL DATA BUS TO DAUGHTER BOARD

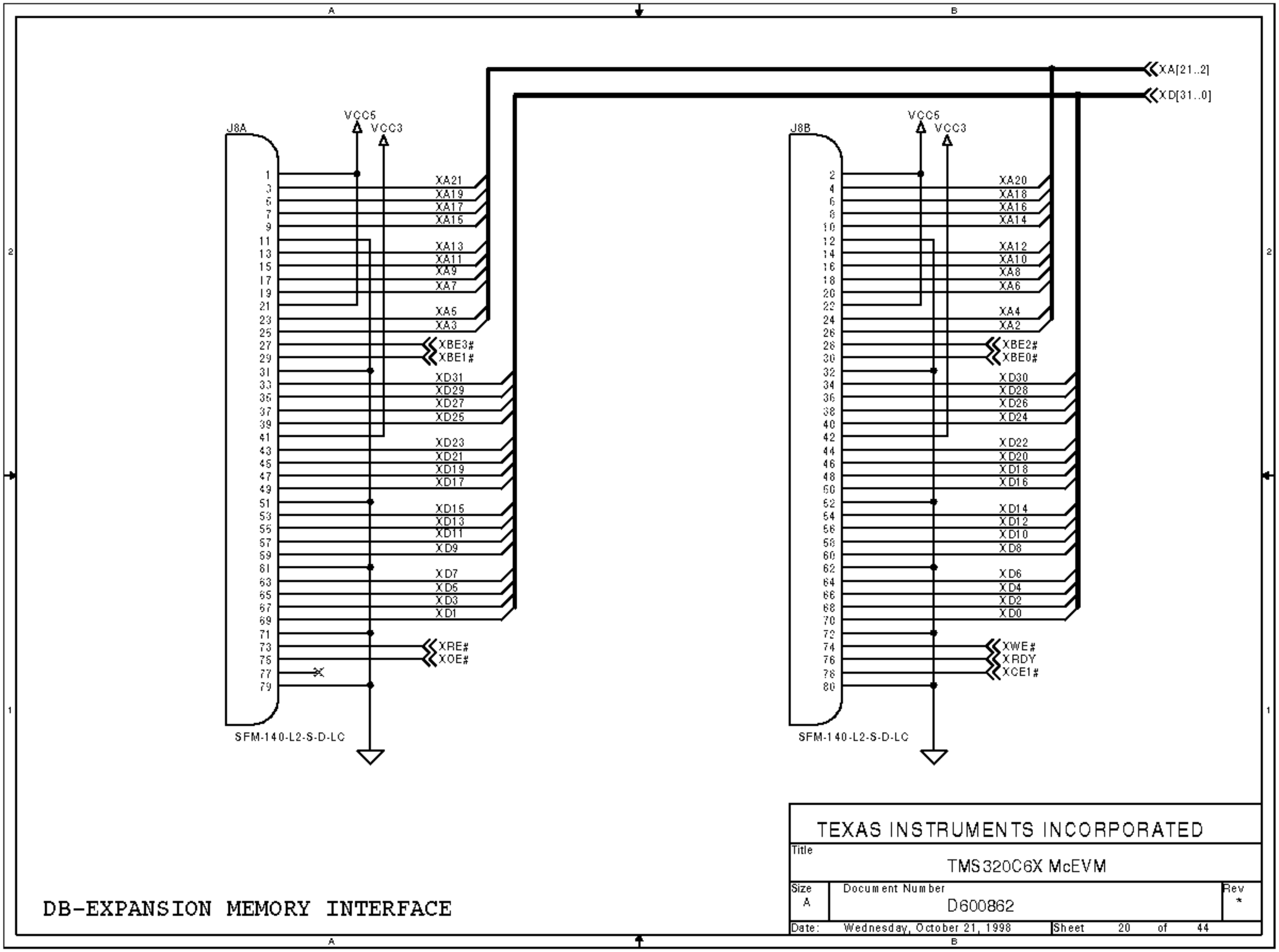
TEXAS INSTRUMENTS INCORPORATED

Title
TMS320C6X McEVM

Size A
Document Number D600862
Rev *

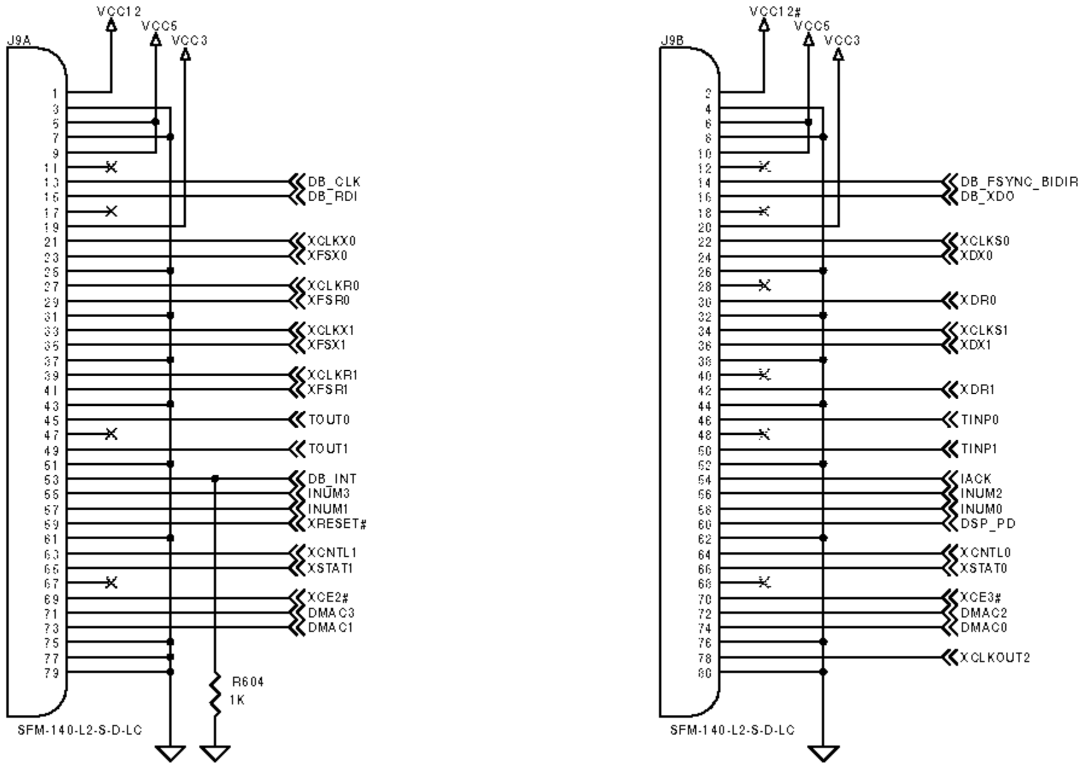
Date: Wednesday, October 21, 1998
Sheet 18 of 44


VCC3
U18
SN74ALVCH16244
GADDR2
GADDR3
GADDR4
GADDR5
GADDR6
GADDR7
GADDR8
GADDR9
GADDR10
GADDR11
GADDR12
GADDR13
GADDR14
GADDR15
GADDR16
GADDR17
XA2
XA3
XA4
XA5
XA6
XA7
XA8
XA9
XA10
XA11
XA12
XA13
XA14
XA15
XA16
XA17
TP1004
120
R613
GADDR[21..2]
U22
SN74ALVCH16244
GADDR18
GADDR19
GADDR20
GADDR21
XA18
XA19
XA20
XA21
XA[21..2]
ARE#
AOE#
AWE#
G_BE0#
G_BE1#
G_BE2#
G_BE3#
CLKOUT2
G_CE1#
G_CE2#
G_CE3#
XRE#
XOE#
XWE#
XBE0#
XBE1#
XBE2#
XBE3#
XCLKOUT2
XCE1#
XCE2#
XCE3#
CE1_DIR
TP1003
120
R620
TP1005
xU500 TC7SHU04F_NU
VCC5
SSOP5
ALTERNATE FOOTPRINT
U500 TC7SHU04F
VCC5
AOE#
SOT23-5
PLACE NEAR U22
DAUGHTERBOARD ADDRESS BUFFERS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 19 of 44



DB-EXPANSION MEMORY INTERFACE

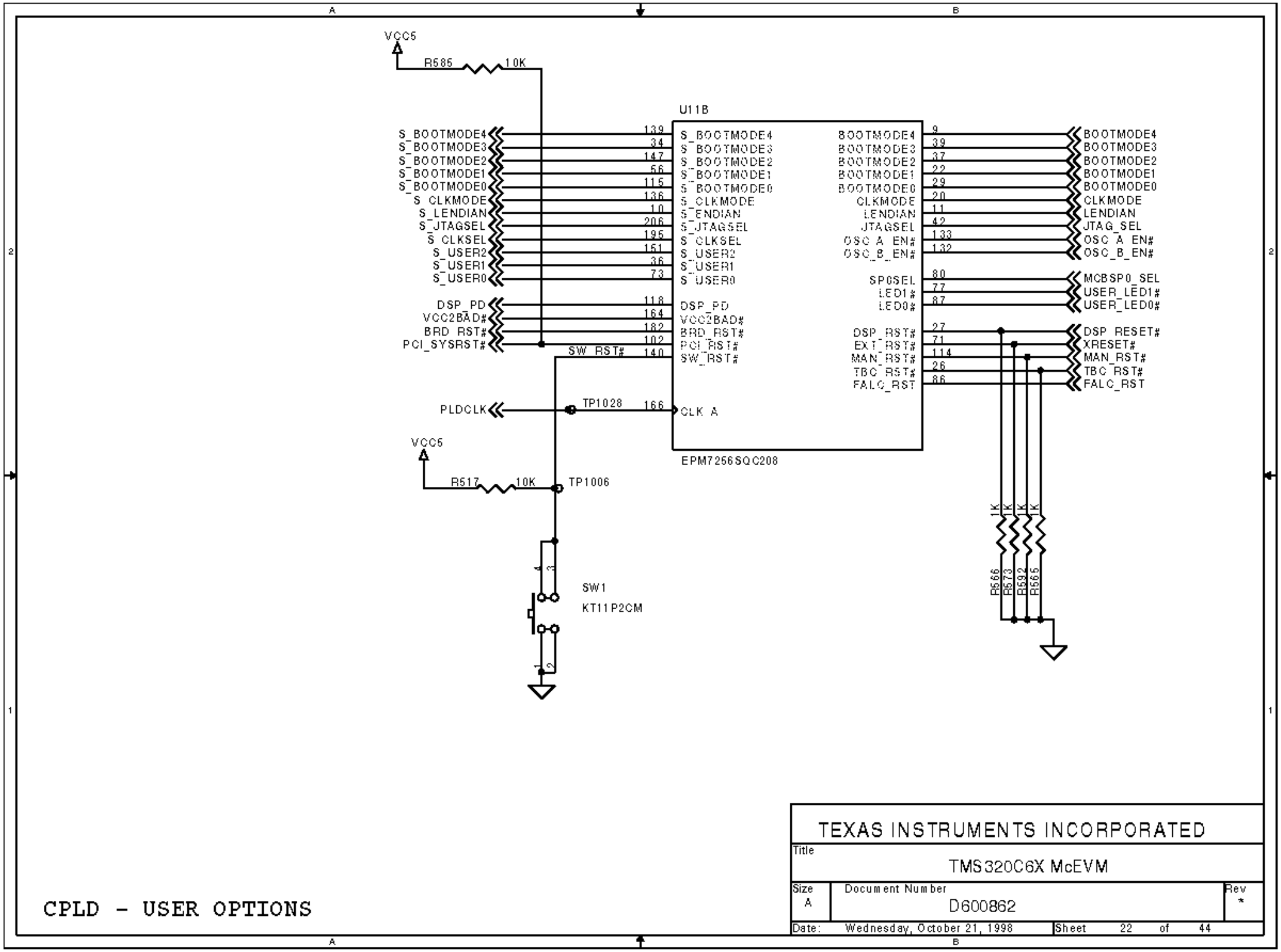
TEXAS INSTRUMENTS INCORPORATED		
Title TMS320C6X McEVM		
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 20 of 44	



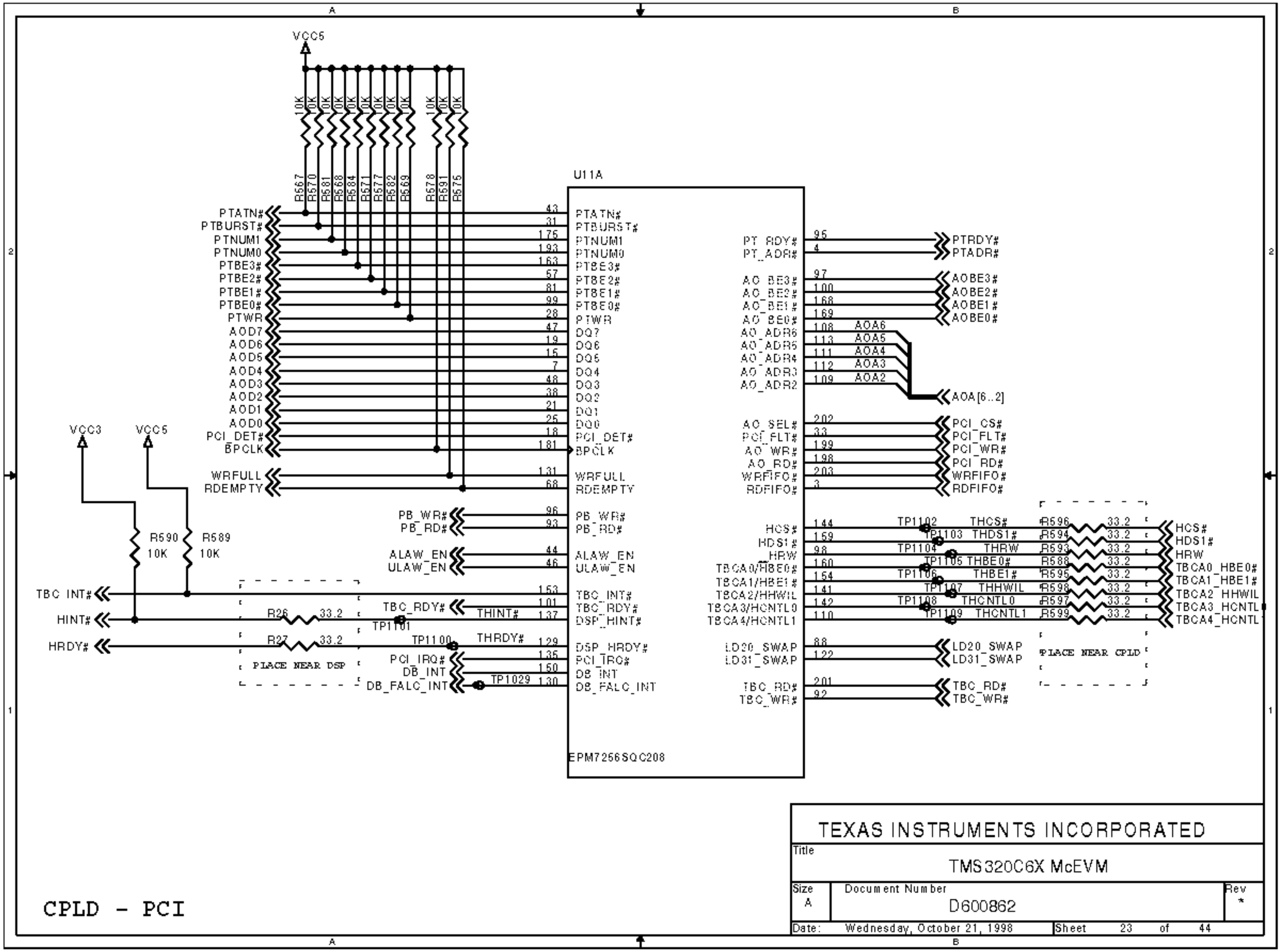
DB-EXPANSION PERIPHERAL INTERFACE

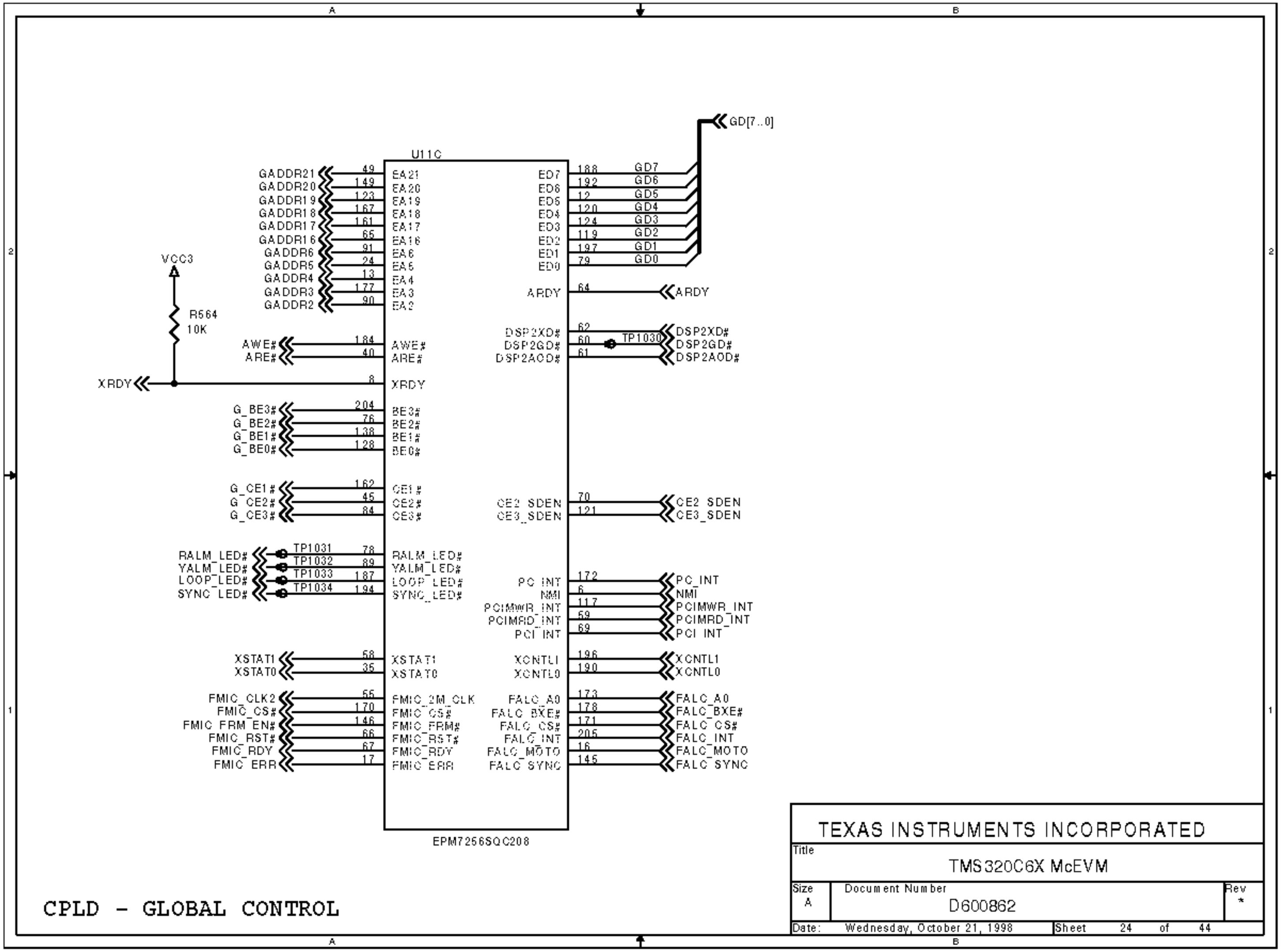
TEXAS INSTRUMENTS INCORPORATED

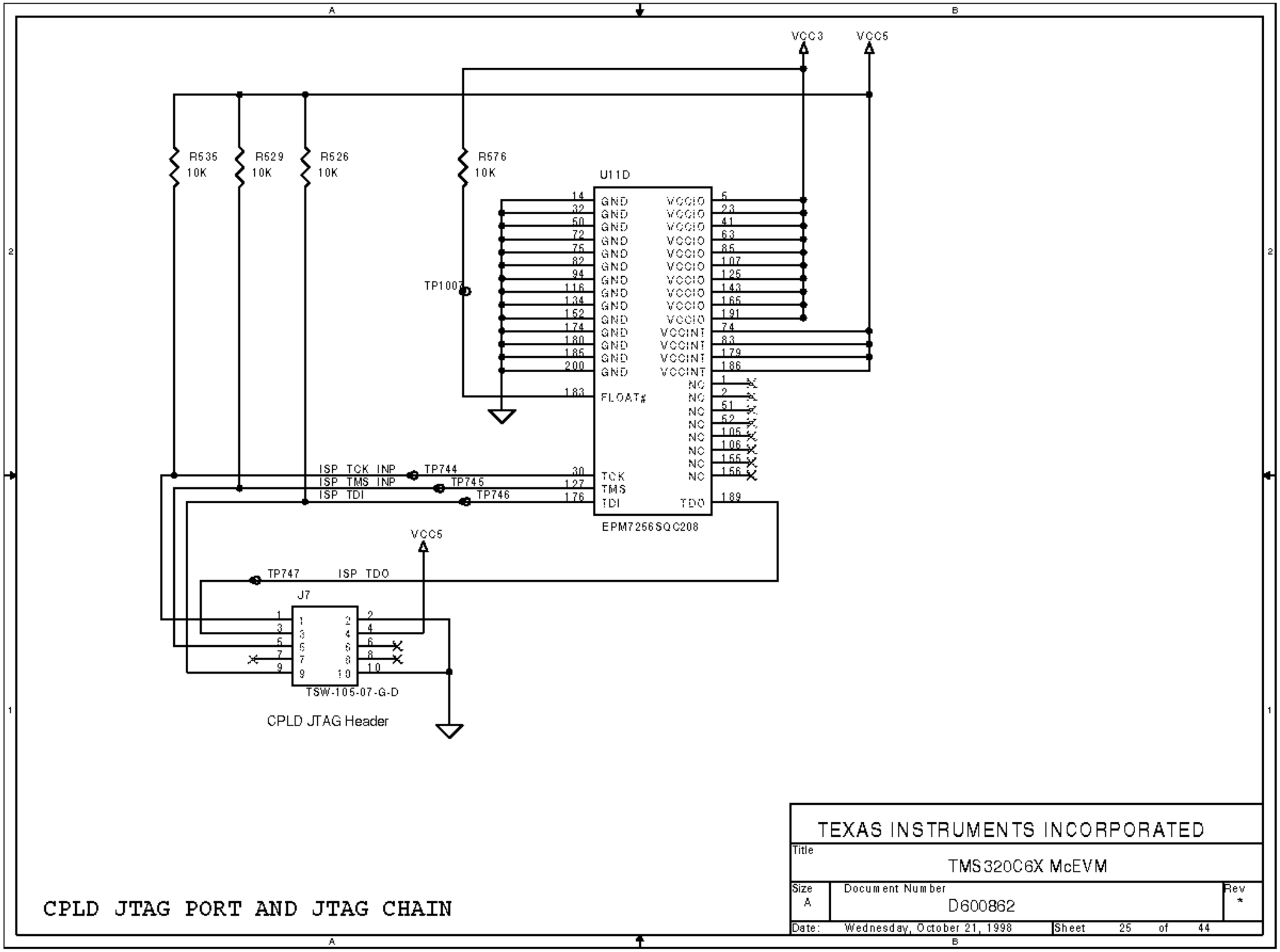
Title TMS320C6X McEVM		
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 21 of 44	



CPLD - USER OPTIONS


VCC5
10K
R567
R570
R581
R568
R584
R571
R577
R582
R569
R578
R591
R575
U11A
PTATN#
PTBURST#
PTNUM1
PTNUM0
PTBE3#
PTBE2#
PTBE1#
PTBE0#
PTWR
AOD7
AOD6
AOD5
AOD4
AOD3
AOD2
AOD1
AOD0
PCI_DET#
BPCLK
WRFULL
RDEMPTY
43
31
175
193
163
57
81
99
28
47
19
15
7
48
38
21
25
18
181
131
68
DQ7
DQ6
DQ5
DQ4
DQ3
DQ2
DQ1
DQ0
PB_WR#
PB_RD#
96
93
ALAW_EN
ULAW_EN
44
46
VCC3
VCC5
R590
10K
R589
10K
TBC_INT#
HINT#
HRDY#
R26
33.2
R27
33.2
PLACE NEAR DSP
TBC_RDY#
THINT#
TP1101
TP1100
THRDY#
PCI_IRQ#
DB_INT
DB_FALC_INT
TP1029
153
101
137
129
135
150
130
DSP_HINT#
DSP_HRDY#
PCI_IRQ#
EPM7256SQC208
PT_RDY#
PT_ADR#
95
4
PTRDY#
PTADR#
AO_BE3#
AO_BE2#
AO_BE1#
AO_BE0#
97
100
168
169
AOBE3#
AOBE2#
AOBE1#
AOBE0#
AO_ADR6
AO_ADR5
AO_ADR4
AO_ADR3
AO_ADR2
108
113
111
112
109
AOA6
AOA5
AOA4
AOA3
AOA2
AOA[6..2]
AO_SEL#
PCI_FLT#
AO_WR#
AO_RD#
WRFIFO#
RDFIFO#
202
33
199
198
203
3
PCI_CS#
PCI_FLT#
PCI_WR#
PCI_RD#
WRFIFO#
RDFIFO#
HCS#
HDS1#
HRW
TBCA0/HBE0#
TBCA1/HBE1#
TBCA2/HHWIL
TBCA3/HCNTL0
TBCA4/HCNTL1
144
159
98
160
154
141
142
110
TP1102
TP1103
TP1104
TP1105
TP1106
TP1107
TP1108
TP1109
THCS#
THDS1#
THRW
THBE0#
THBE1#
THHWIL
THCNTL0
THCNTL1
R596
R594
R593
R588
R595
R598
R597
R599
33.2
HCS#
HDS1#
HRW
TBCA0_HBE0#
TBCA1_HBE1#
TBCA2_HHWIL
TBCA3_HCNTL0
TBCA4_HCNTL1
PLACE NEAR CPLD
LD20_SWAP
LD31_SWAP
88
122
TBC_RD#
TBC_WR#
201
92
A
B
2
1
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date:
Wednesday, October 21, 1998
Sheet
23
of
44
CPLD - PCI


GD[7..0]
U11C
GADDR21 49 EA21
GADDR20 149 EA20
GADDR19 123 EA19
GADDR18 167 EA18
GADDR17 161 EA17
GADDR16 65 EA16
GADDR6 91 EA6
GADDR5 24 EA5
GADDR4 13 EA4
GADDR3 177 EA3
GADDR2 90 EA2
VCC3
R564
10K
AWE# 184 AWE#
ARE# 40 ARE#
XRDY 8 XRDY
G_BE3# 204 BE3#
G_BE2# 76 BE2#
G_BE1# 138 BE1#
G_BE0# 128 BE0#
G_CE1# 162 CE1#
G_CE2# 45 CE2#
G_CE3# 84 CE3#
RALM_LED# TP1031 78 RALM_LED#
YALM_LED# TP1032 89 YALM_LED#
LOOP_LED# TP1033 187 LOOP_LED#
SYNC_LED# TP1034 194 SYNC_LED#
XSTAT1 58 XSTAT1
XSTAT0 35 XSTAT0
FMIC_CLK2 55 FMIC_2M_CLK
FMIC_CS# 170 FMIC_CS#
FMIC_FRM_EN# 146 FMIC_FRM#
FMIC_RST# 66 FMIC_RST#
FMIC_RDY 67 FMIC_RDY
FMIC_ERR 17 FMIC_ERR
ED7 188 GD7
ED6 192 GD6
ED5 12 GD5
ED4 120 GD4
ED3 124 GD3
ED2 119 GD2
ED1 197 GD1
ED0 79 GD0
ARDY 64 ARDY
DSP2XD# 62 DSP2XD#
DSP2GD# 60 TP1030 DSP2GD#
DSP2AOD# 61 DSP2AOD#
CE2_SDEN 70 CE2_SDEN
CE3_SDEN 121 CE3_SDEN
PC_INT 172 PC_INT
NMI 6 NMI
PCIMWR_INT 117 PCIMWR_INT
PCIMRD_INT 59 PCIMRD_INT
PCI_INT 69 PCI_INT
XCNTL1 196 XCNTL1
XCNTL0 190 XCNTL0
FALC_A0 173 FALC_A0
FALC_BXE# 178 FALC_BXE#
FALC_CS# 171 FALC_CS#
FALC_INT 205 FALC_INT
FALC_MOTO 16 FALC_MOTO
FALC_SYNC 145 FALC_SYNC
EPM7256SQC208
A
B
1
2
CPLD - GLOBAL CONTROL
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 24 of 44



CPLD JTAG PORT AND JTAG CHAIN

TEXAS INSTRUMENTS INCORPORATED

Title: TMS320C6X McEVM

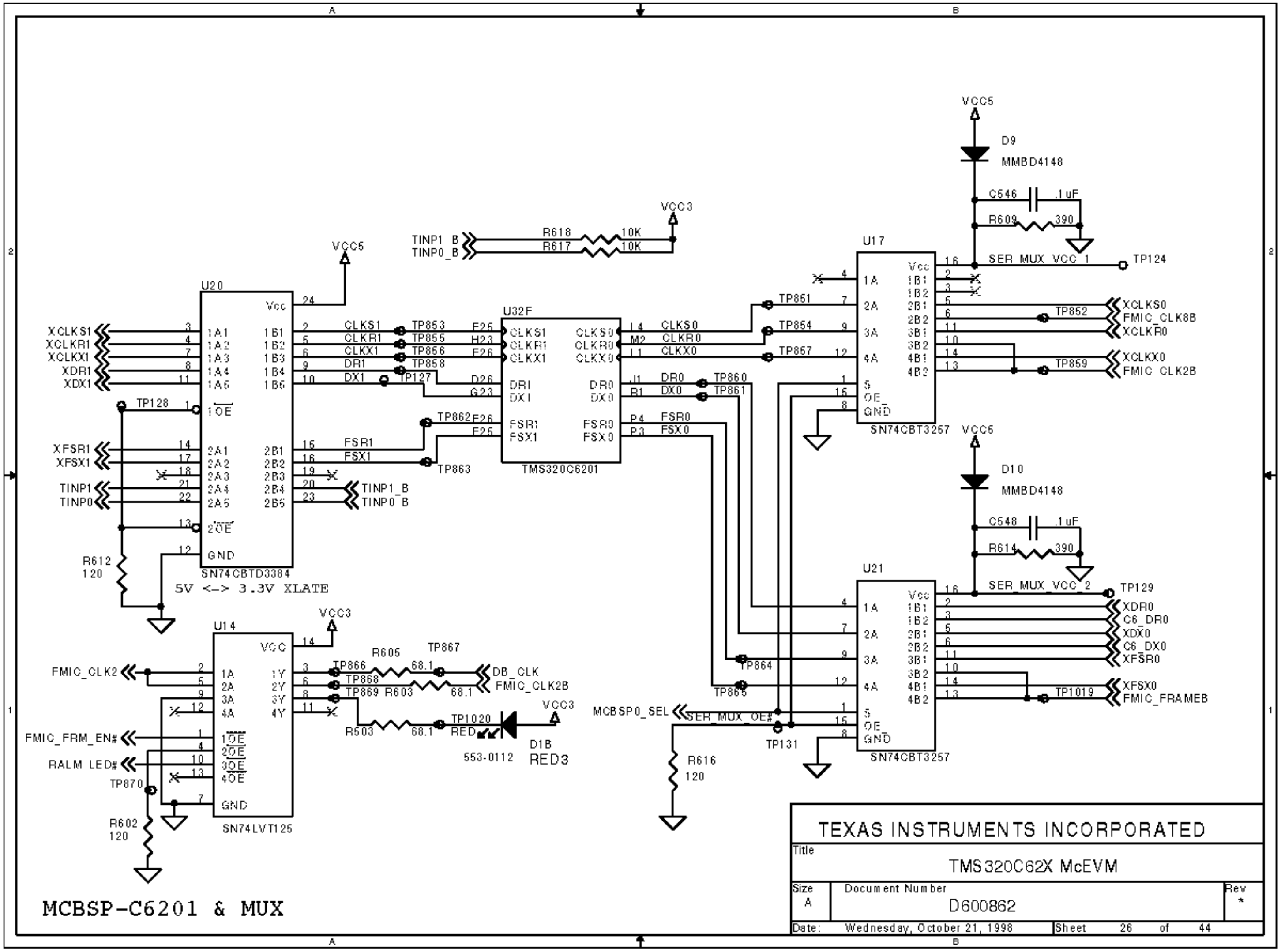
Size: A

Document Number: D600862

Rev: *

Date: Wednesday, October 21, 1998

Sheet 25 of 44

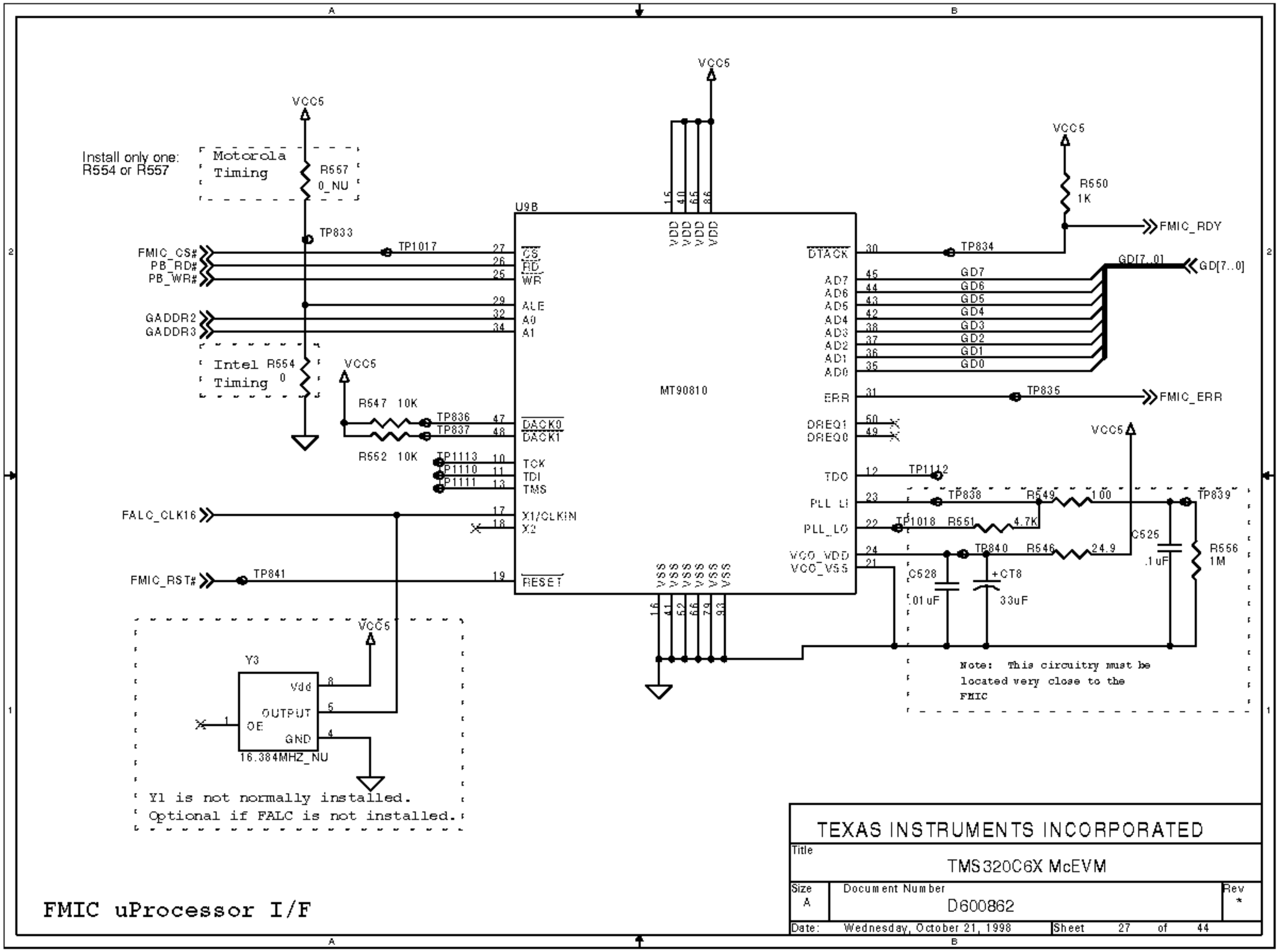


TEXAS INSTRUMENTS INCORPORATED

Title
TMS320C62X McEVM

Size	Document Number	Rev
A	D600862	*

Date: Wednesday, October 21, 1998 Sheet 26 of 44



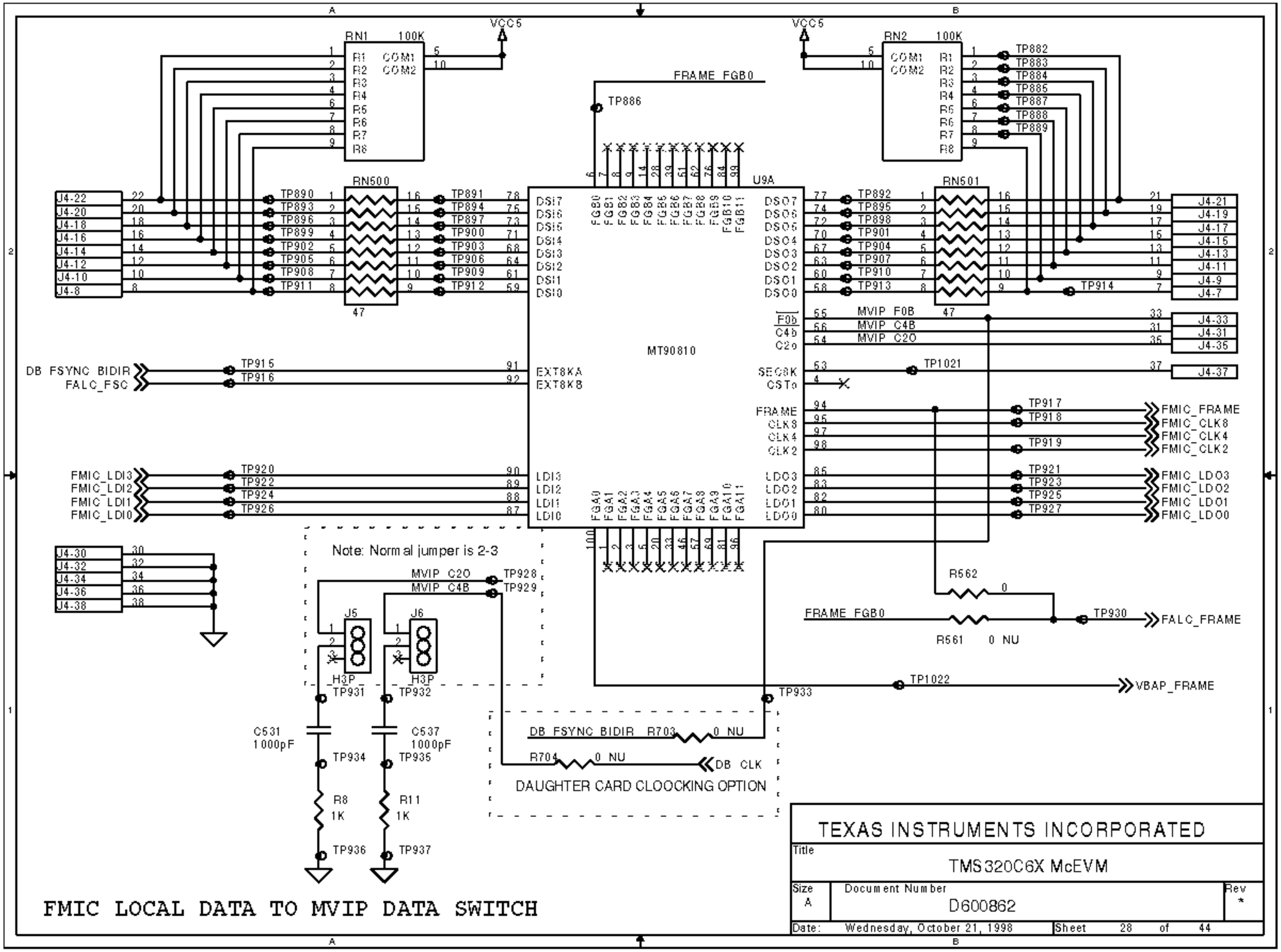
FMIC uProcessor I/F

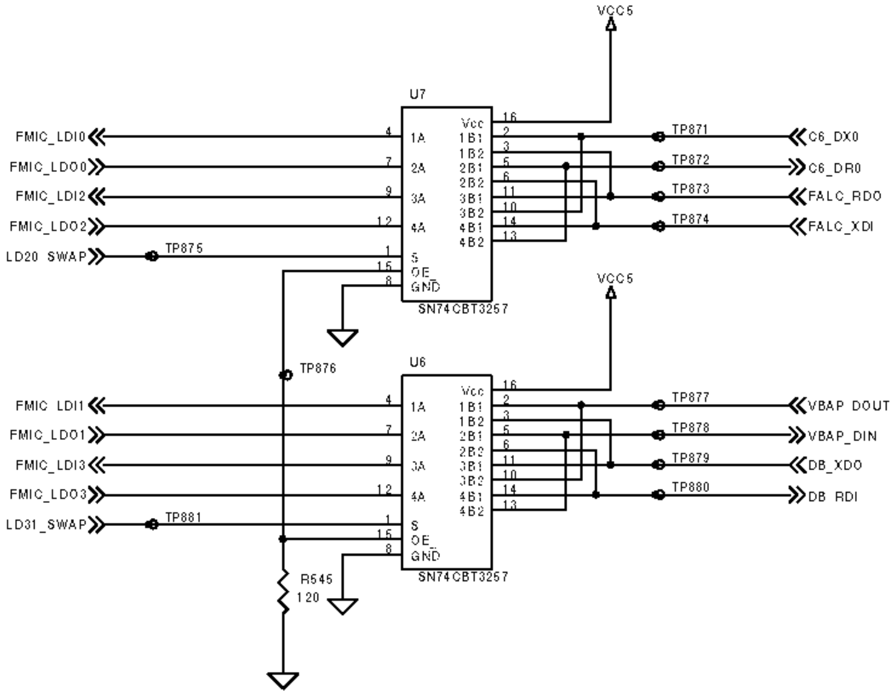
TEXAS INSTRUMENTS INCORPORATED

Title: TMS320C6X McEVM

Size: A | Document Number: D600862 | Rev: *

Date: Wednesday, October 21, 1998 | Sheet 27 of 44

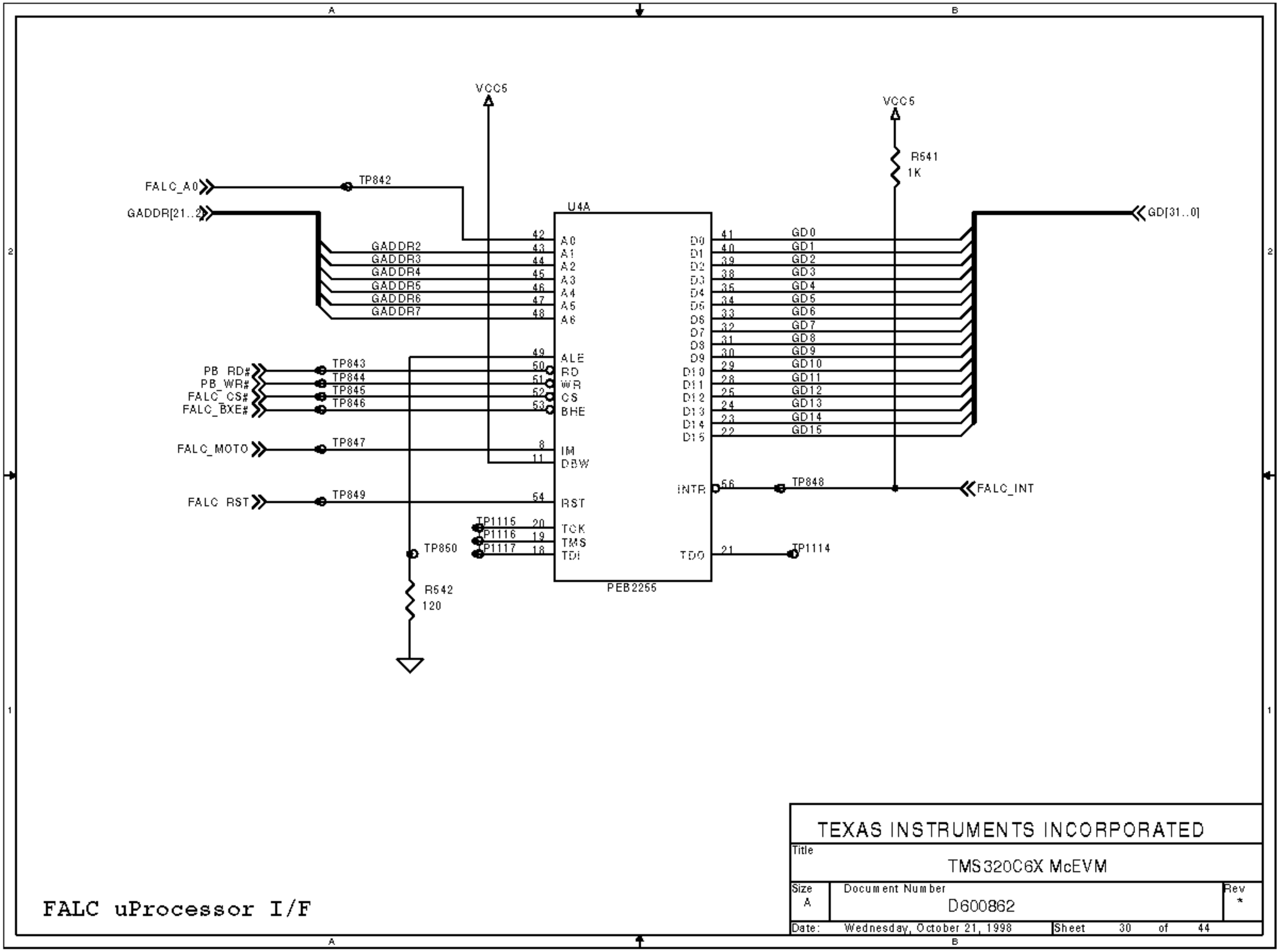

RN1 100K
R1 R2 R3 R4 R5 R6 R7 R8
COM1 COM2
VCC5
RN2 100K
TP882
TP883
TP884
TP885
TP887
TP888
TP889
FRAME_FGB0
TP886
U9A
MT90810
RN500
RN501
47
J4-22
J4-20
J4-18
J4-16
J4-14
J4-12
J4-10
J4-8
TP890
TP893
TP896
TP899
TP902
TP905
TP908
TP911
TP891
TP894
TP897
TP900
TP903
TP906
TP909
TP912
DSi7
DSi6
DSi5
DSi4
DSi3
DSi2
DSi1
DSi0
FGB0
FGB1
FGB2
FGB3
FGB4
FGB5
FGB6
FGB7
FGB8
FGB9
FGB10
FGB11
DSo7
DSo6
DSo5
DSo4
DSo3
DSo2
DSo1
DSo0
TP892
TP895
TP898
TP901
TP904
TP907
TP910
TP913
TP914
J4-21
J4-19
J4-17
J4-15
J4-13
J4-11
J4-9
J4-7
F0b
C4b
C2o
MVIP_F0B
MVIP_C4B
MVIP_C2O
J4-33
J4-31
J4-35
SEC8K
CSTo
TP1021
J4-37
DB_FSYNC_BIDIR
FALC_FSC
TP915
TP916
EXT8KA
EXT8KB
FRAME
CLK8
CLK4
CLK2
TP917
TP918
TP919
FMIC_FRAME
FMIC_CLK8
FMIC_CLK4
FMIC_CLK2
FMIC_LDI3
FMIC_LDI2
FMIC_LDI1
FMIC_LDI0
TP920
TP922
TP924
TP926
LDI3
LDI2
LDI1
LDI0
FGA0
FGA1
FGA2
FGA3
FGA4
FGA5
FGA6
FGA7
FGA8
FGA9
FGA10
FGA11
LDO3
LDO2
LDO1
LDO0
TP921
TP923
TP925
TP927
FMIC_LDO3
FMIC_LDO2
FMIC_LDO1
FMIC_LDO0
J4-30
J4-32
J4-34
J4-36
J4-38
Note: Normal jumper is 2-3
MVIP_C2O
MVIP_C4B
TP928
TP929
J5
J6
H3P
TP931
TP932
C531
1000pF
C537
1000pF
TP934
TP935
R8
1K
R11
1K
TP936
TP937
R562
0
FRAME_FGB0
R561 0 NU
TP930
FALC_FRAME
TP1022
VBAP_FRAME
TP933
DB_FSYNC_BIDIR
R703
0 NU
R704
0 NU
DB_CLK
DAUGHTER CARD CLOOCKING OPTION
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 28 of 44
FMIC LOCAL DATA TO MVIP DATA SWITCH

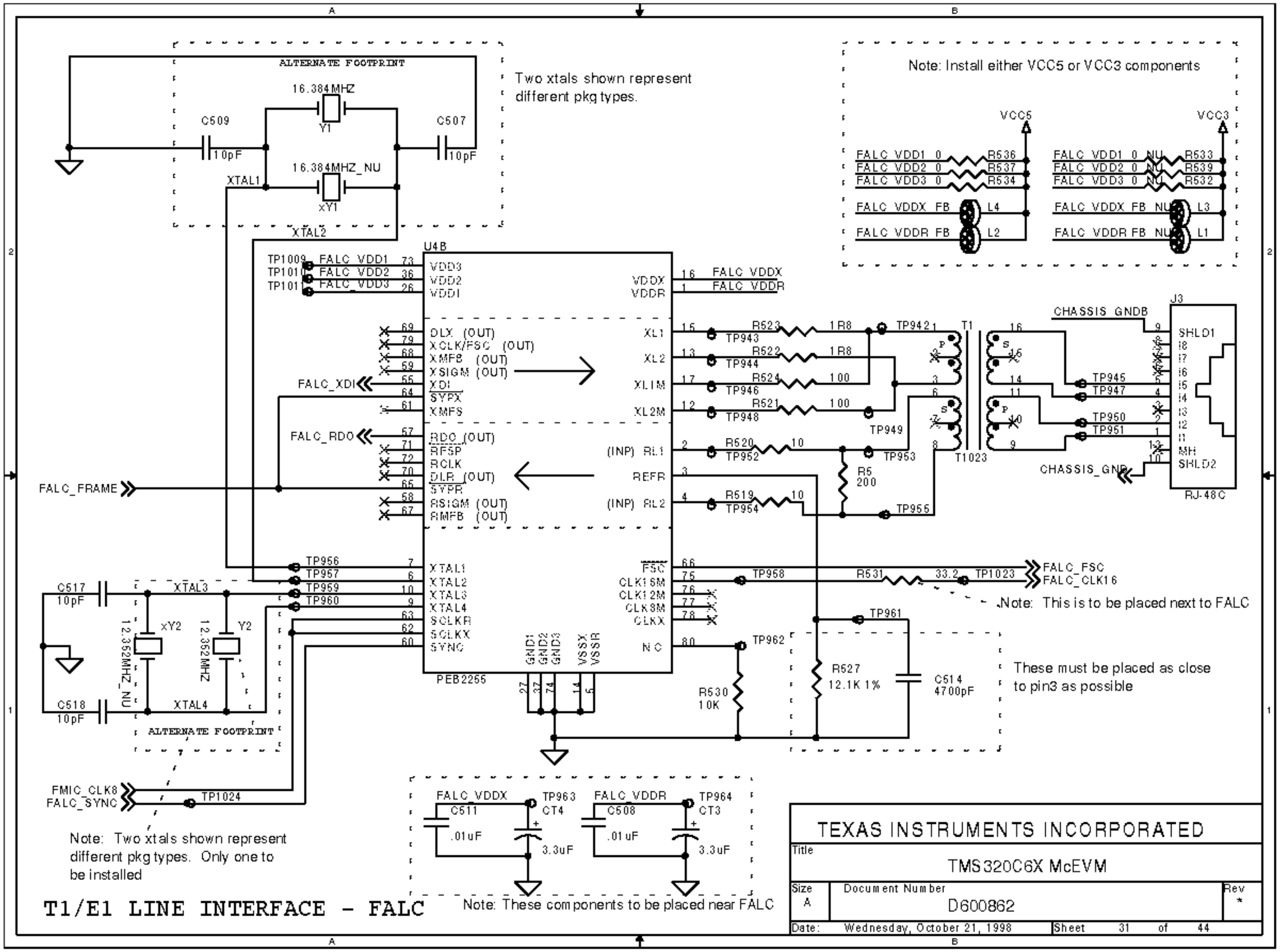


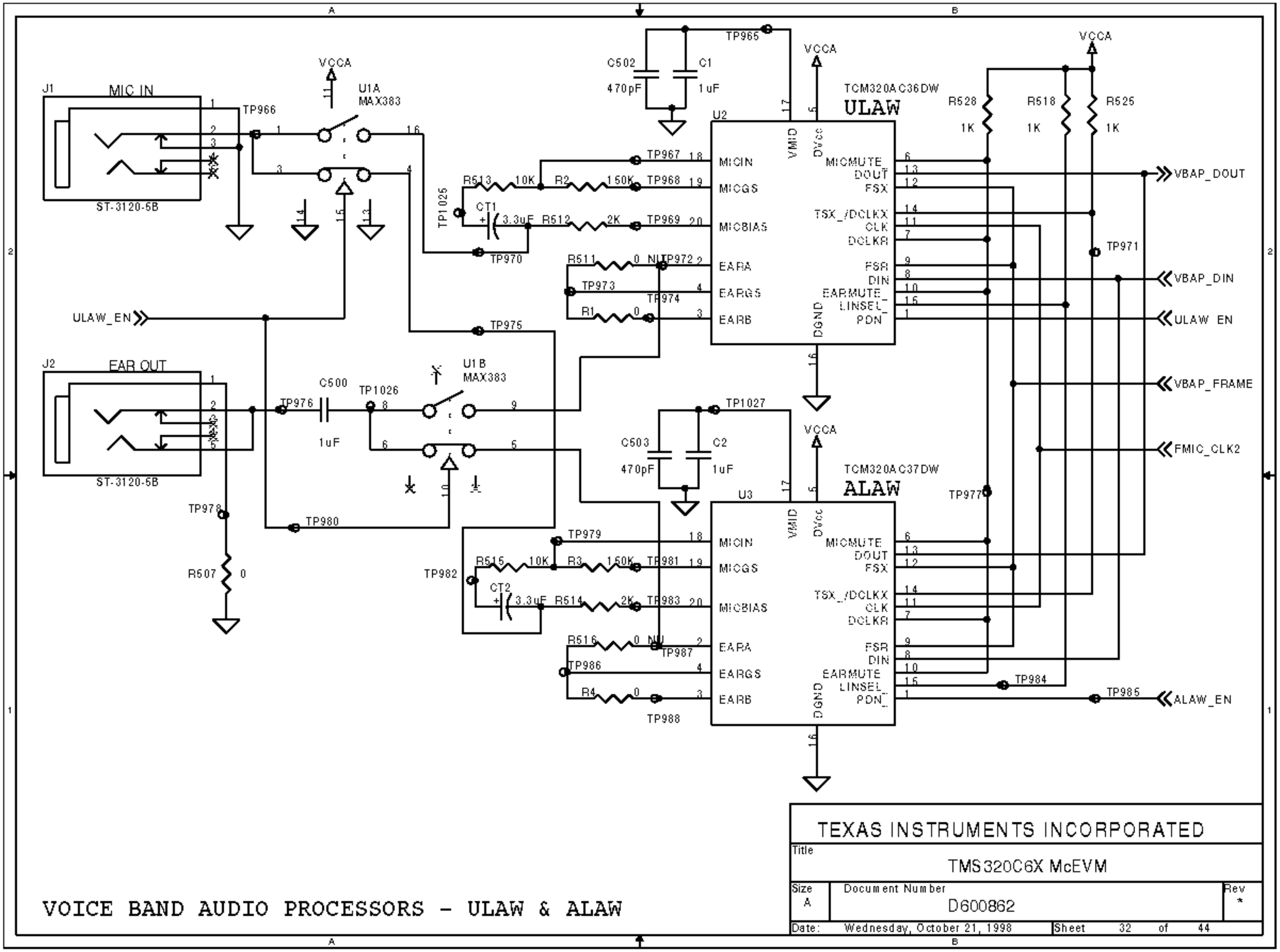
FMIC SERIAL PORT MUXES

TEXAS INSTRUMENTS INCORPORATED

Title TMS320C62X McEVM		
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 29 of 44	


A
B
2
1
VCC5
VCC5
R541
1K
FALC_A0
TP842
GADDR[21..2]
U4A
GD[31..0]
GADDR2
GADDR3
GADDR4
GADDR5
GADDR6
GADDR7
42 A0
43 A1
44 A2
45 A3
46 A4
47 A5
48 A6
49 ALE
50 RD
51 WR
52 CS
53 BHE
8 IM
11 DBW
54 RST
20 TCK
19 TMS
18 TDI
D0 41 GD0
D1 40 GD1
D2 39 GD2
D3 38 GD3
D4 35 GD4
D5 34 GD5
D6 33 GD6
D7 32 GD7
D8 31 GD8
D9 30 GD9
D10 29 GD10
D11 28 GD11
D12 25 GD12
D13 24 GD13
D14 23 GD14
D15 22 GD15
INTR 56
TP848
FALC_INT
TDO 21
TP1114
PEB2255
PB_RD#
PB_WR#
FALC_CS#
FALC_BXE#
TP843
TP844
TP845
TP846
FALC_MOTO
TP847
FALC_RST
TP849
TP1115
TP1116
TP1117
TP850
R542
120
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 30 of 44
FALC uProcessor I/F

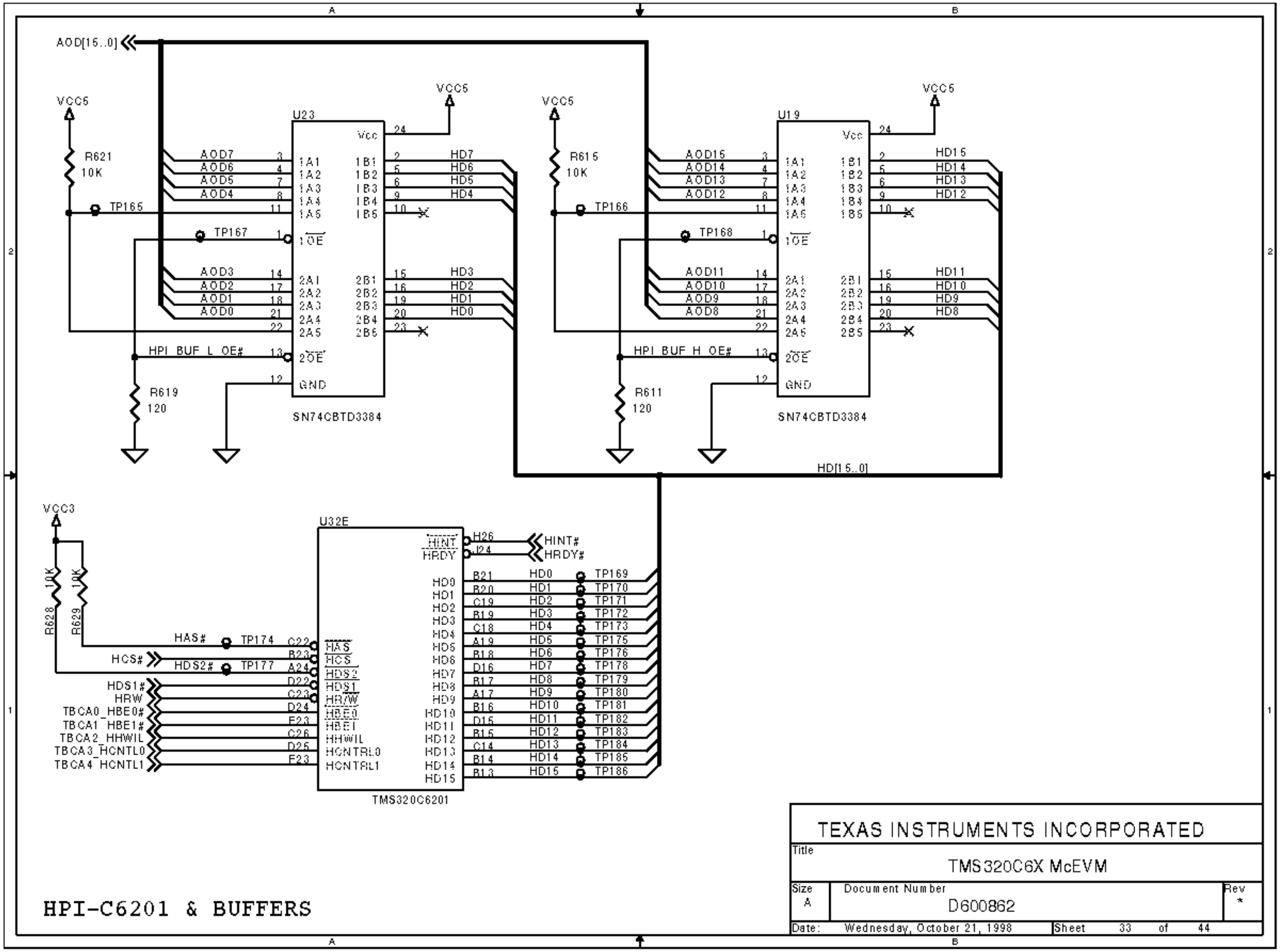

ALTERNATE FOOTPRINT
16.384MHZ
Y1
16.384MHZ_NU
xY1
C509
10pF
C507
10pF
XTAL1
XTAL2
Two xtals shown represent different pkg types.
Note: Install either VCC5 or VCC3 components
VCC5
VCC3
FALC_VDD1 0 R536
FALC_VDD2 0 R537
FALC_VDD3 0 R534
FALC_VDDX_FB L4
FALC_VDDR_FB L2
FALC_VDD1 0_NU R533
FALC_VDD2 0_NU R539
FALC_VDD3 0_NU R532
FALC_VDDX_FB_NU L3
FALC_VDDR_FB_NU L1
U4B
TP1009 FALC_VDD1 73 VDD3
TP1010 FALC_VDD2 36 VDD2
TP1011 FALC_VDD3 26 VDD1
VDDX 16 FALC_VDDX
VDDR 1 FALC_VDDR
DLX (OUT)
XCLK/FSC (OUT)
XMFB (OUT)
XSIGM (OUT)
XDI
SYPX
XMFS
FALC_XDI
FALC_RDO
RDO (OUT)
RFSP
RCLK
DLR (OUT)
SYPR
RSIGM (OUT)
RMFB (OUT)
FALC_FRAME
XL1 XL2 XLIM XL2M
(INP) RL1
REFR
(INP) RL2
R523 1R8 TP943
R522 1R8 TP944
R524 100 TP946
R521 100 TP948
R520 10 TP952
R519 10 TP954
R5 200
TP942 TP949 TP953 TP955
T1 T1023
CHASSIS_GNDB
CHASSIS_GND
J3
SHLD1 MH SHLD2
RJ-48C
TP945 TP947 TP950 TP951
TP956 XTAL1
TP957 XTAL2
TP959 XTAL3
TP960 XTAL4
SCLKR
SCLKX
SYNC
GND1 GND2 GND3 VSSX VSSR
PEB2255
FSC
CLK16M
CLK12M
CLK8M
CLKX
NC
TP958
R531 33.2 TP1023
FALC_FSC
FALC_CLK16
Note: This is to be placed next to FALC
TP961
TP962
R530 10K
R527 12.1K 1%
C514 4700pF
These must be placed as close to pin3 as possible
C517 10pF
C518 10pF
XTAL3
XTAL4
xY2 12.352MHZ_NU
Y2 12.352MHZ
ALTERNATE FOOTPRINT
FMIC_CLK8
FALC_SYNC
TP1024
Note: Two xtals shown represent different pkg types. Only one to be installed
FALC_VDDX TP963 C511 CT4 .01uF 3.3uF
FALC_VDDR TP964 C508 CT3 .01uF 3.3uF
Note: These components to be placed near FALC
T1/E1 LINE INTERFACE - FALC
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size A
Document Number D600862
Rev *
Date: Wednesday, October 21, 1998
Sheet 31 of 44

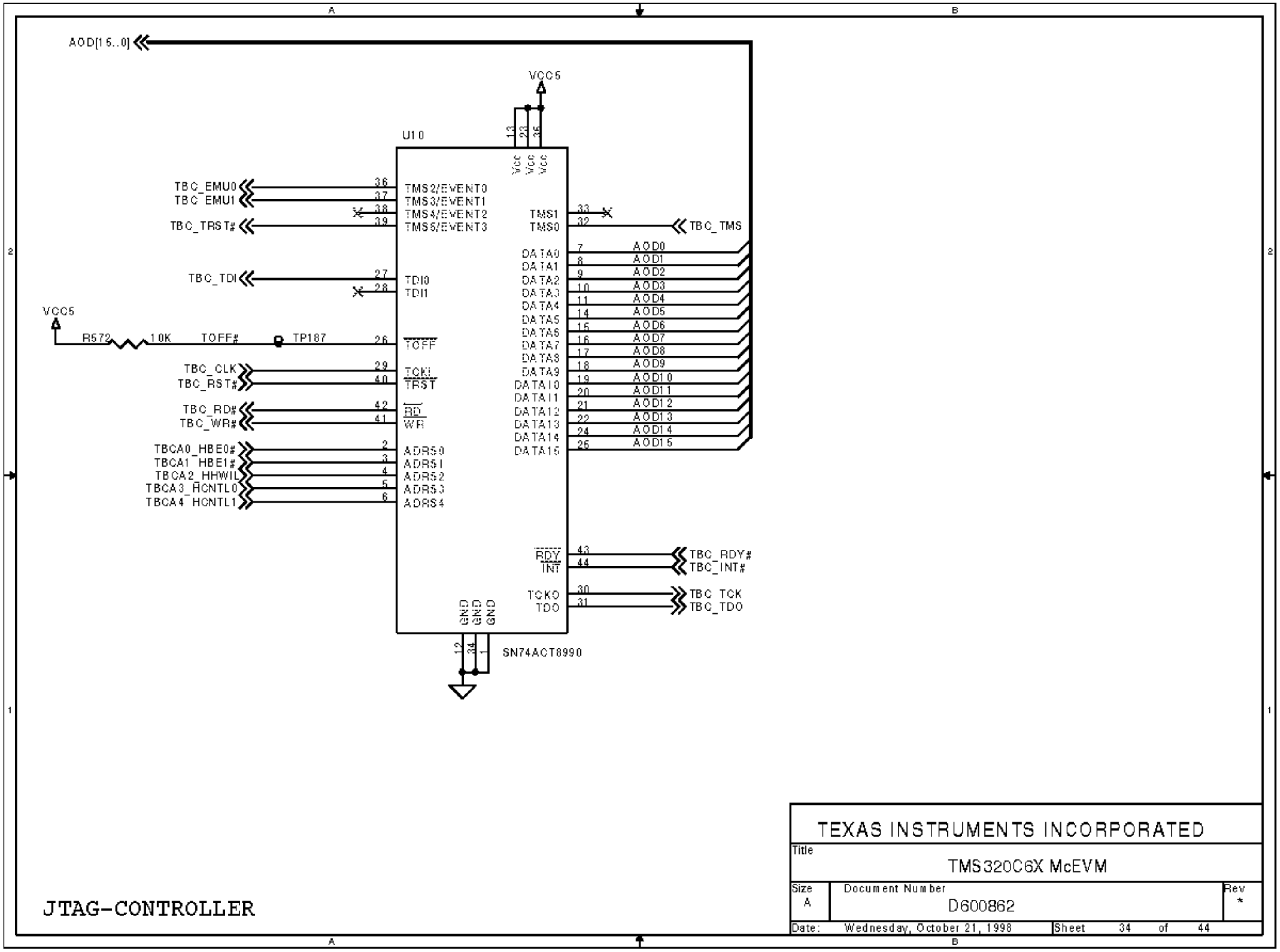


VOICE BAND AUDIO PROCESSORS - ULAW & ALAW

TEXAS INSTRUMENTS INCORPORATED

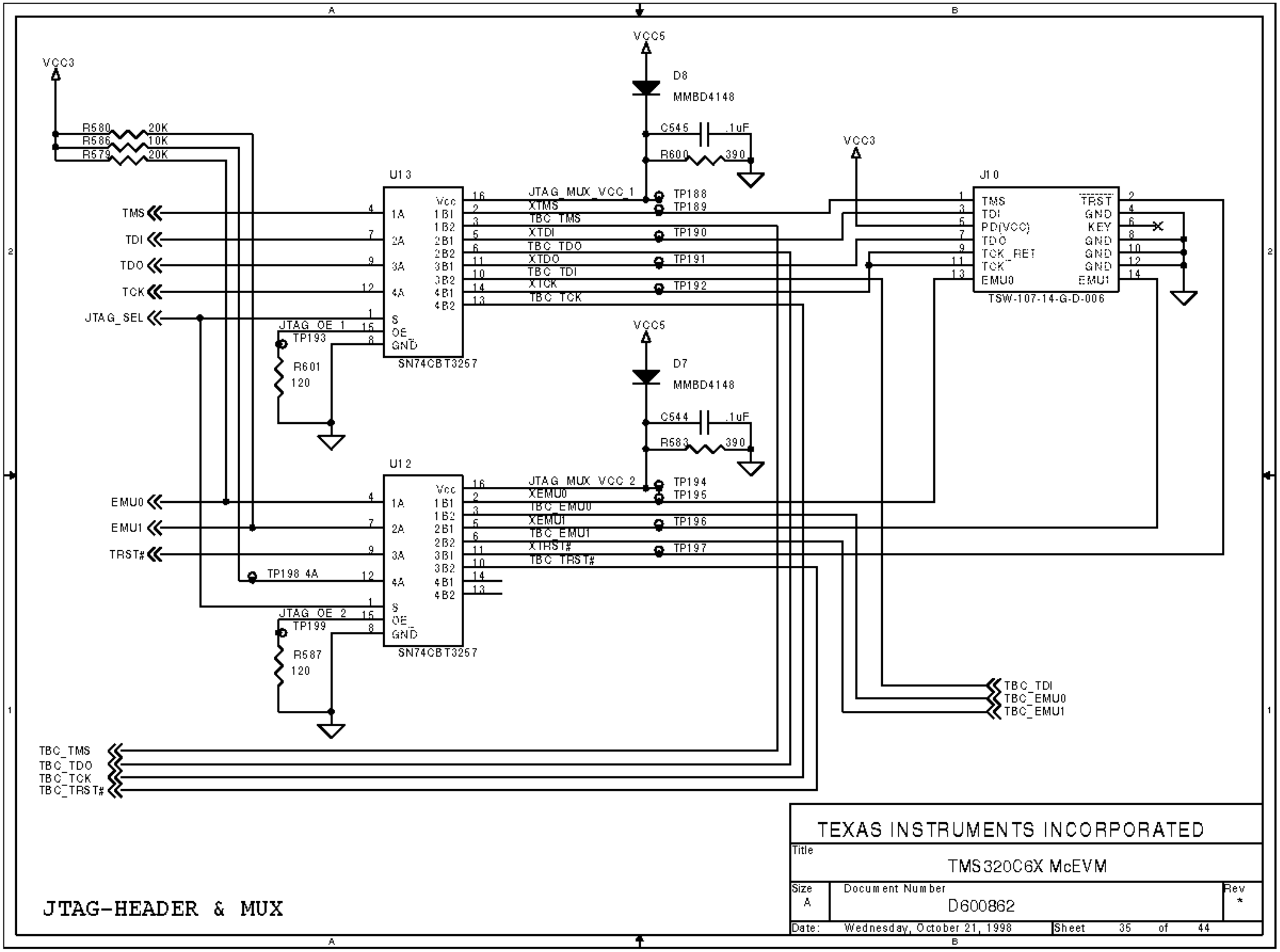
Title	TMS320C6X McEVM	
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 32 of 44	

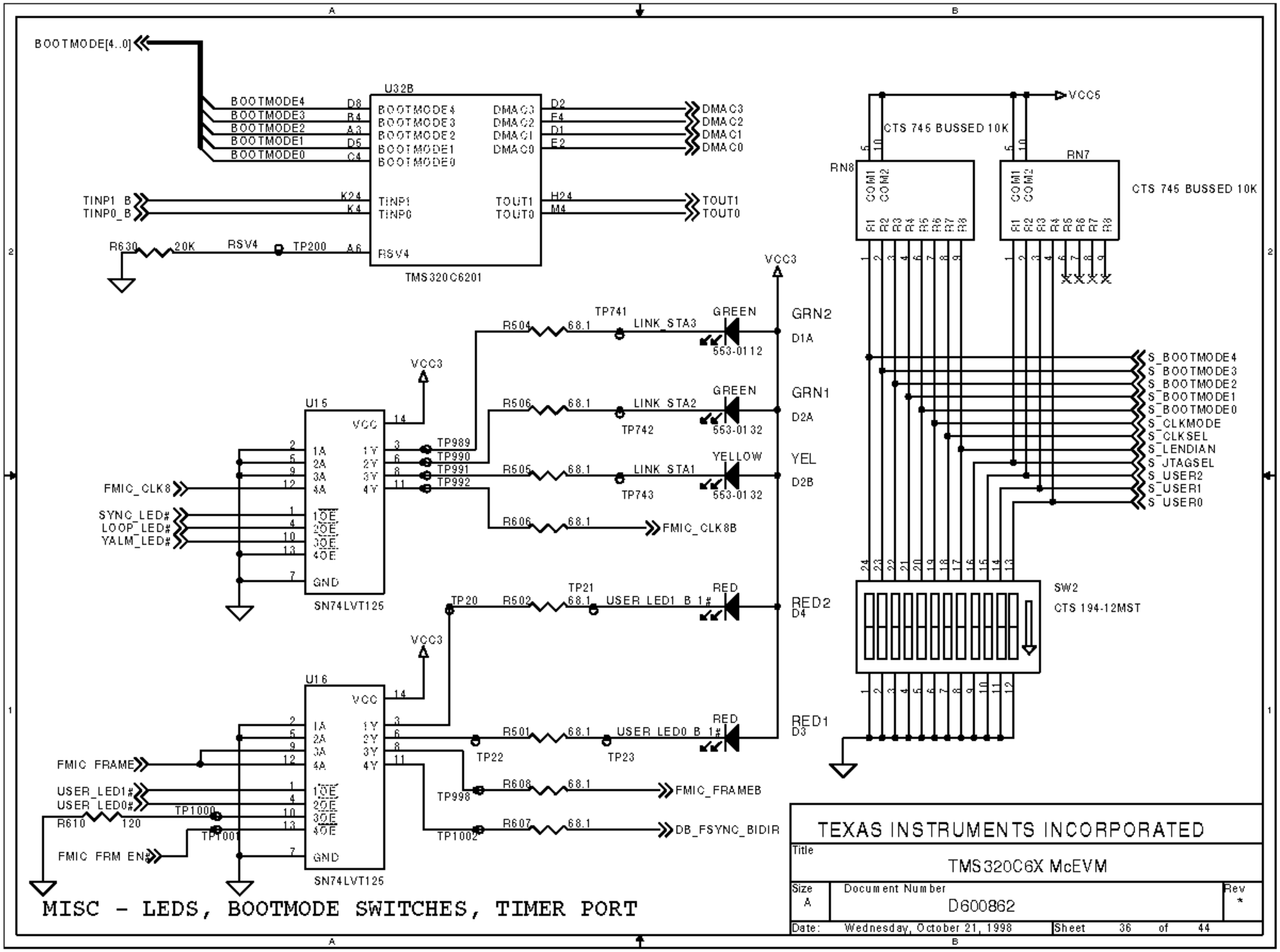

AOD[15..0]
VCC5
R621
10K
TP165
U23
Vcc
24
AOD7
AOD6
AOD5
AOD4
1A1
1A2
1A3
1A4
1A5
1B1
1B2
1B3
1B4
1B5
HD7
HD6
HD5
HD4
TP167
1OE
AOD3
AOD2
AOD1
AOD0
2A1
2A2
2A3
2A4
2A5
2B1
2B2
2B3
2B4
2B5
HD3
HD2
HD1
HD0
HPI_BUF_L_OE#
2OE
GND
R619
120
SN74CBTD3384
R615
10K
TP166
U19
AOD15
AOD14
AOD13
AOD12
HD15
HD14
HD13
HD12
TP168
AOD11
AOD10
AOD9
AOD8
HD11
HD10
HD9
HD8
HPI_BUF_H_OE#
R611
120
SN74CBTD3384
HD[15..0]
VCC3
R628
10K
R629
10K
U32E
HINT
HRDY
H26
J24
HINT#
HRDY#
HAS#
TP174
HCS#
HDS2#
TP177
HDS1#
HRW
TBCA0_HBE0#
TBCA1_HBE1#
TBCA2_HHWIL
TBCA3_HCNTL0
TBCA4_HCNTL1
C22
B23
A24
D22
C23
D24
E23
C26
D25
F23
HAS
HCS
HDS2
HDS1
HR/W
HBE0
HBE1
HHWIL
HCNTRL0
HCNTRL1
HD0
HD1
HD2
HD3
HD4
HD5
HD6
HD7
HD8
HD9
HD10
HD11
HD12
HD13
HD14
HD15
B21
B20
C19
B19
C18
A19
B18
D16
B17
A17
B16
D15
B15
C14
B14
B13
TP169
TP170
TP171
TP172
TP173
TP175
TP176
TP178
TP179
TP180
TP181
TP182
TP183
TP184
TP185
TP186
TMS320C6201
HPI-C6201 & BUFFERS
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 33 of 44

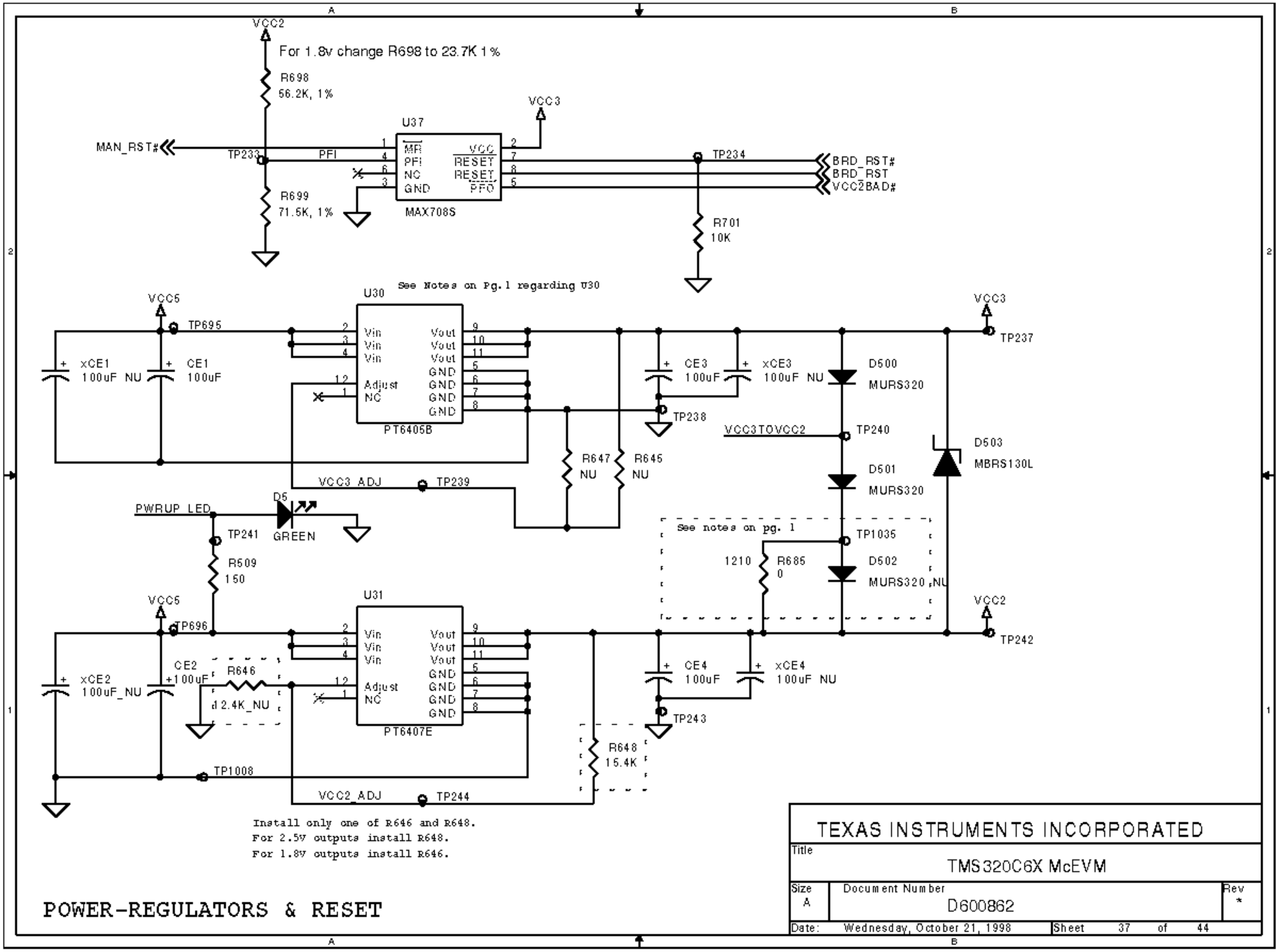


TEXAS INSTRUMENTS INCORPORATED

Title	TMS320C6X McEVM		
Size A	Document Number D600862		Rev *
Date: Wednesday, October 21, 1998	Sheet 34 of 44		


VCC3
R580 20K
R586 10K
R579 20K
TMS
TDI
TDO
TCK
JTAG_SEL
EMU0
EMU1
TRST#
TP198 4A
U13
U12
1A
2A
3A
4A
S
OE_
GND
Vcc
1B1
1B2
2B1
2B2
3B1
3B2
4B1
4B2
SN74CBT3257
JTAG_OE_1
TP193
R601
120
JTAG_OE_2
TP199
R587
120
VCC5
D8
MMBD4148
C545 .1uF
R600 390
D7
MMBD4148
C544 .1uF
R583 390
JTAG_MUX_VCC_1
XTMS
TBC_TMS
XTDI
TBC_TDO
XTDO
TBC_TDI
XTCK
TBC_TCK
TP188
TP189
TP190
TP191
TP192
JTAG_MUX_VCC_2
XEMU0
TBC_EMU0
XEMU1
TBC_EMU1
XTRST#
TBC_TRST#
TP194
TP195
TP196
TP197
J10
TMS
TDI
PD(VCC)
TDO
TCK_RET
TCK
EMU0
TRST
GND
KEY
GND
GND
GND
EMU1
TSW-107-14-G-D-006
TBC_TDI
TBC_EMU0
TBC_EMU1
TBC_TMS
TBC_TDO
TBC_TCK
TBC_TRST#
JTAG-HEADER & MUX
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 35 of 44
A
B
1
2


BOOTMODE[4..0]
U32B
BOOTMODE4
BOOTMODE3
BOOTMODE2
BOOTMODE1
BOOTMODE0
D8
B4
A3
D5
C4
DMAC3
DMAC2
DMAC1
DMAC0
D2
F4
D1
E2
TINP1_B
TINP0_B
K24
K4
TINP1
TINP0
TOUT1
TOUT0
H24
M4
R630
20K
RSV4
TP200
A6
TMS320C6201
VCC5
CTS 745 BUSSED 10K
RN8
RN7
COM1
COM2
R1
R2
R3
R4
R5
R6
R7
R8
VCC3
TP741
R504
68.1
LINK_STA3
GREEN
GRN2
D1A
553-0112
R506
LINK_STA2
GRN1
D2A
TP742
553-0132
U15
VCC
1A
2A
3A
4A
1Y
2Y
3Y
4Y
1OE
2OE
3OE
4OE
GND
TP989
TP990
TP991
TP992
R505
LINK_STA1
YELLOW
YEL
D2B
TP743
FMIC_CLK8
SYNC_LED#
LOOP_LED#
YALM_LED#
R606
FMIC_CLK8B
SN74LVT125
S_BOOTMODE4
S_BOOTMODE3
S_BOOTMODE2
S_BOOTMODE1
S_BOOTMODE0
S_CLKMODE
S_CLKSEL
S_LENDIAN
S_JTAGSEL
S_USER2
S_USER1
S_USER0
SW2
CTS 194-12MST
TP20
R502
TP21
USER_LED1_B_1#
RED
RED2
D4
U16
FMIC_FRAME
R501
USER_LED0_B_1#
RED1
D3
TP22
TP23
USER_LED1#
USER_LED0#
R610
120
TP1000
TP1001
R608
FMIC_FRAMEB
TP998
R607
DB_FSYNC_BIDIR
TP1002
FMIC_FRM_EN#
MISC - LEDS, BOOTMODE SWITCHES, TIMER PORT
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 36 of 44



Install only one of R646 and R648.
For 2.5V outputs install R648.
For 1.8V outputs install R646.

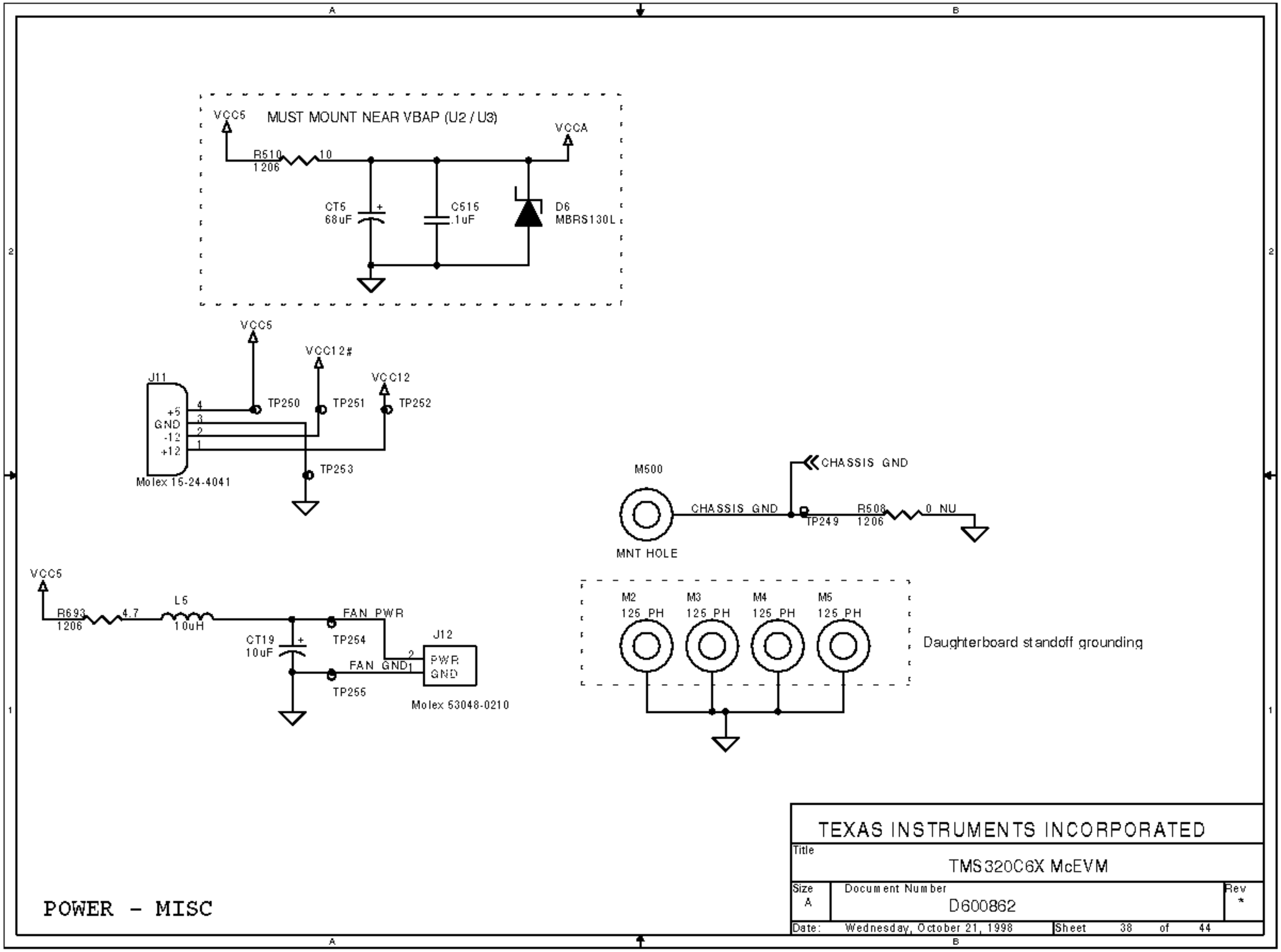
POWER-REGULATORS & RESET

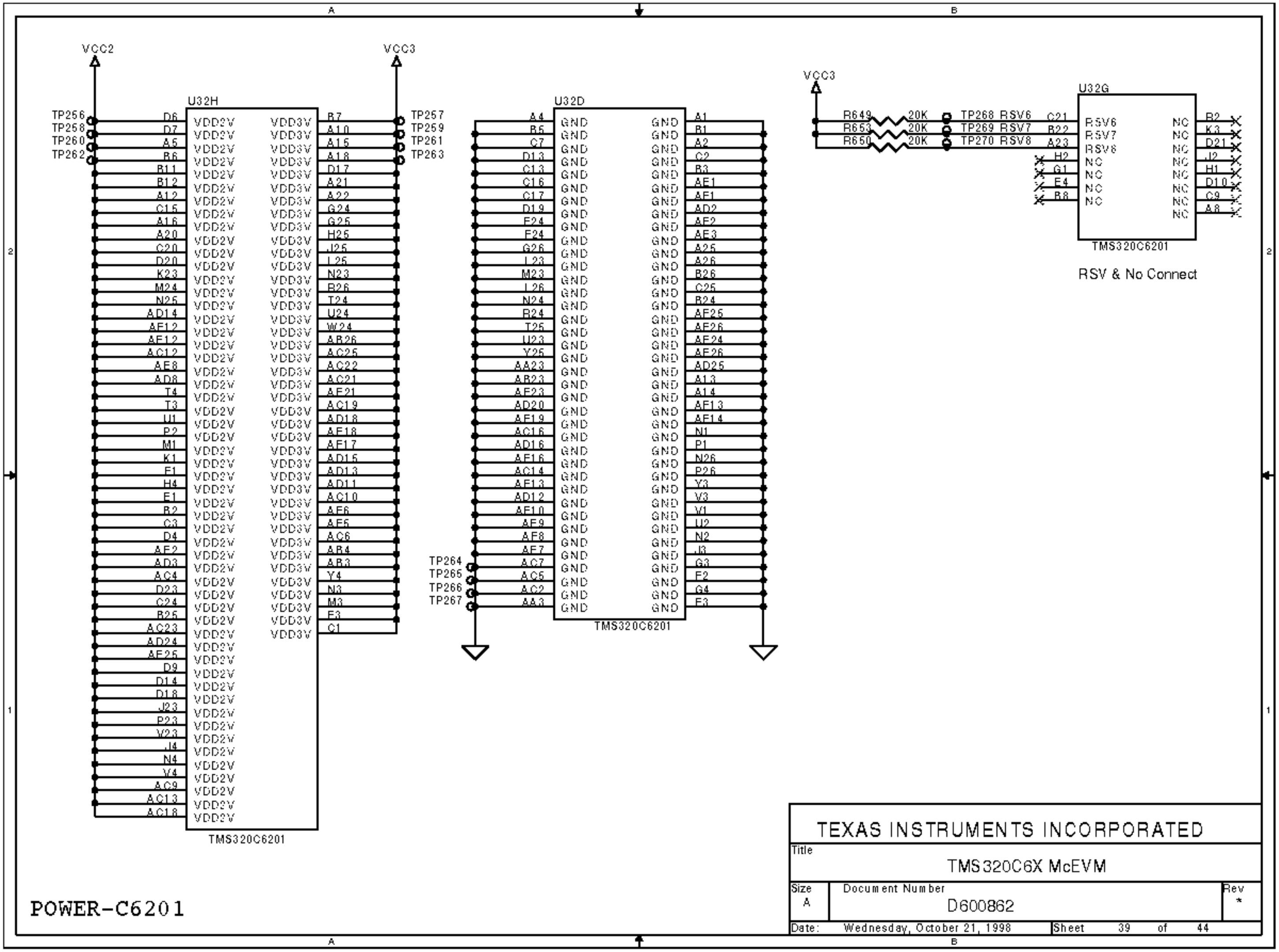
TEXAS INSTRUMENTS INCORPORATED

Title: TMS320C6X McEVM

Size: A
Document Number: D600862
Rev: *

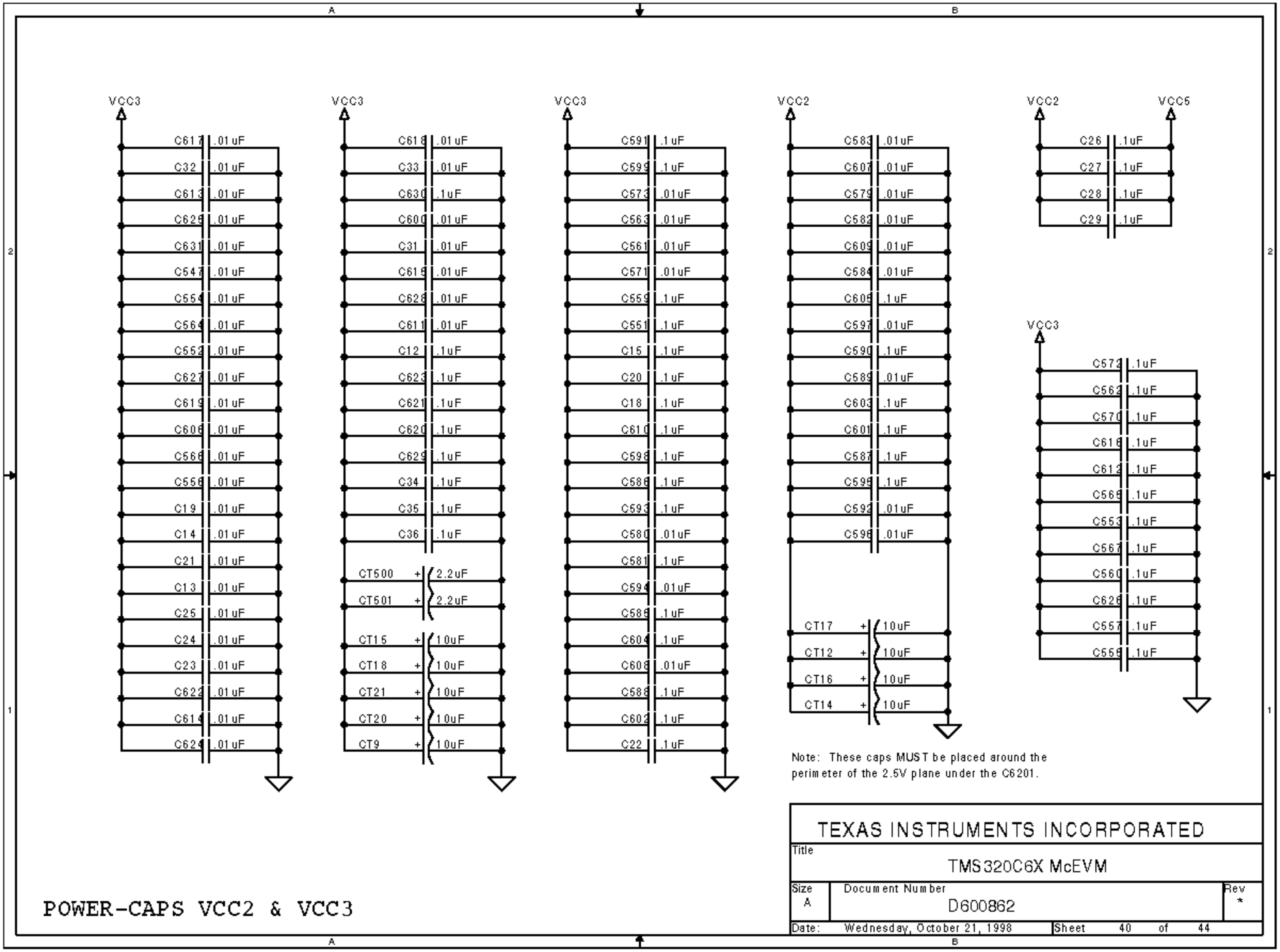
Date: Wednesday, October 21, 1998
Sheet 37 of 44


A
B
2
1
MUST MOUNT NEAR VBAP (U2 / U3)
VCC5
VCCA
R510
1206
10
CT5
68uF
C515
.1uF
D6
MBRS130L
VCC5
VCC12#
VCC12
J11
+5
GND
-12
+12
4
3
2
1
TP250
TP251
TP252
TP253
Molex 15-24-4041
M500
MNT HOLE
CHASSIS GND
CHASSIS GND
TP249
R508
1206
0 NU
VCC5
R693
1206
4.7
L5
10uH
CT19
10uF
FAN PWR
FAN GND
TP254
TP255
J12
PWR
GND
2
1
Molex 53048-0210
M2
125 PH
M3
125 PH
M4
125 PH
M5
125 PH
Daughterboard standoff grounding
POWER - MISC
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 38 of 44


A
B
2
1
VCC2
VCC3
U32H
TP256
TP258
TP260
TP262
D6 D7 A5 B6 B11 B12 A12 C15 A16 A20 C20 D20 K23 M24 N25 AD14 AF12 AE12 AC12 AE8 AD8 T4 T3 U1 P2 M1 K1 F1 H4 E1 B2 C3 D4 AE2 AD3 AC4 D23 C24 B25 AC23 AD24 AE25 D9 D14 D18 J23 P23 V23 J4 N4 V4 AC9 AC13 AC18
VDD2V
VDD3V
B7 A10 A15 A18 D17 A21 A22 G24 G25 H25 J25 L25 N23 R26 T24 U24 W24 AB26 AC25 AC22 AC21 AE21 AC19 AD18 AE18 AF17 AD15 AD13 AD11 AC10 AE6 AE5 AC6 AB4 AB3 Y4 N3 M3 F3 C1
TP257
TP259
TP261
TP263
TMS320C6201
U32D
A4 B5 C7 D13 C13 C16 C17 D19 E24 F24 G26 L23 M23 L26 N24 R24 T25 U23 Y25 AA23 AB23 AE23 AD20 AE19 AC16 AD16 AE16 AC14 AE13 AD12 AE10 AE9 AE8 AE7 AC7 AC5 AC2 AA3
GND
A1 B1 A2 C2 B3 AE1 AF1 AD2 AF2 AE3 A25 A26 B26 C25 B24 AE25 AE26 AF24 AF26 AD25 A13 A14 AF13 AF14 N1 P1 N26 P26 Y3 V3 V1 U2 N2 J3 G3 F2 G4 E3
TP264
TP265
TP266
TP267
TMS320C6201
VCC3
U32G
R649 20K TP268 RSV6 C21
R653 20K TP269 RSV7 B22
R650 20K TP270 RSV8 A23
H2 G1 E4 B8
RSV6 RSV7 RSV8 NC NC NC NC
NC
B2 K3 D21 J2 H1 D10 C9 A8
TMS320C6201
RSV & No Connect
POWER-C6201

TEXAS INSTRUMENTS INCORPORATED

Title TMS320C6X McEVM		
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 39 of 44	

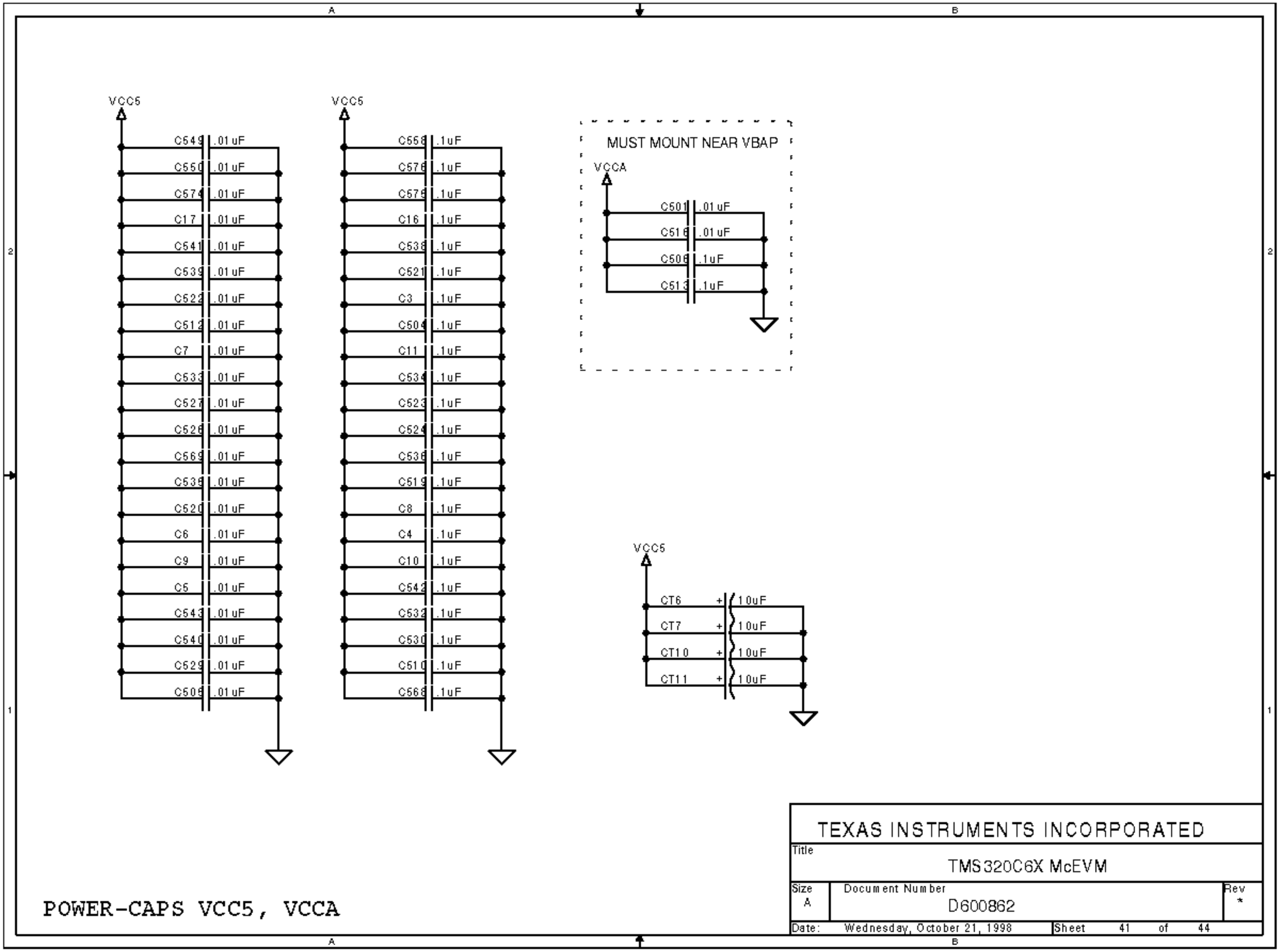


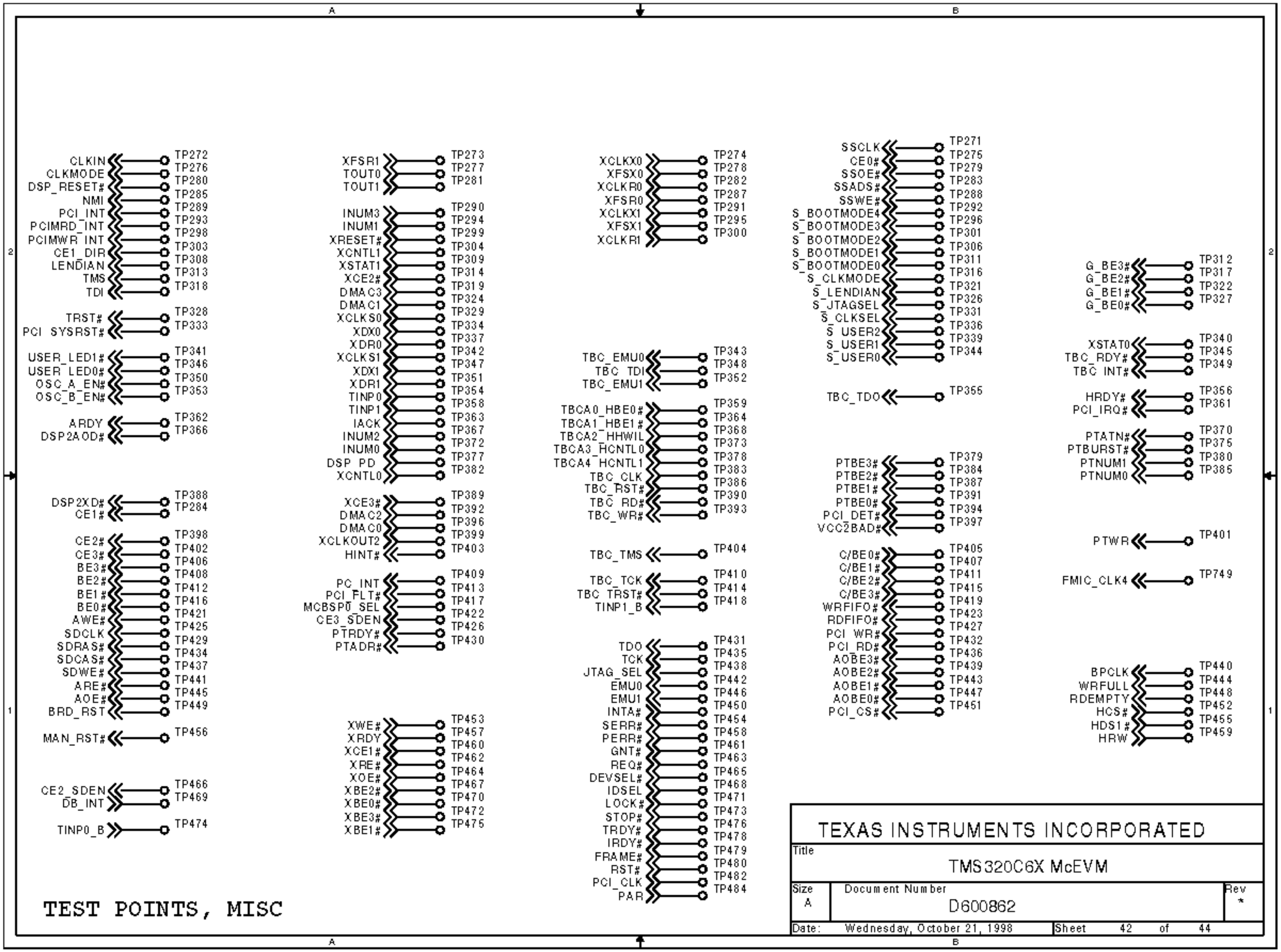
Note: These caps MUST be placed around the perimeter of the 2.5V plane under the C6201.

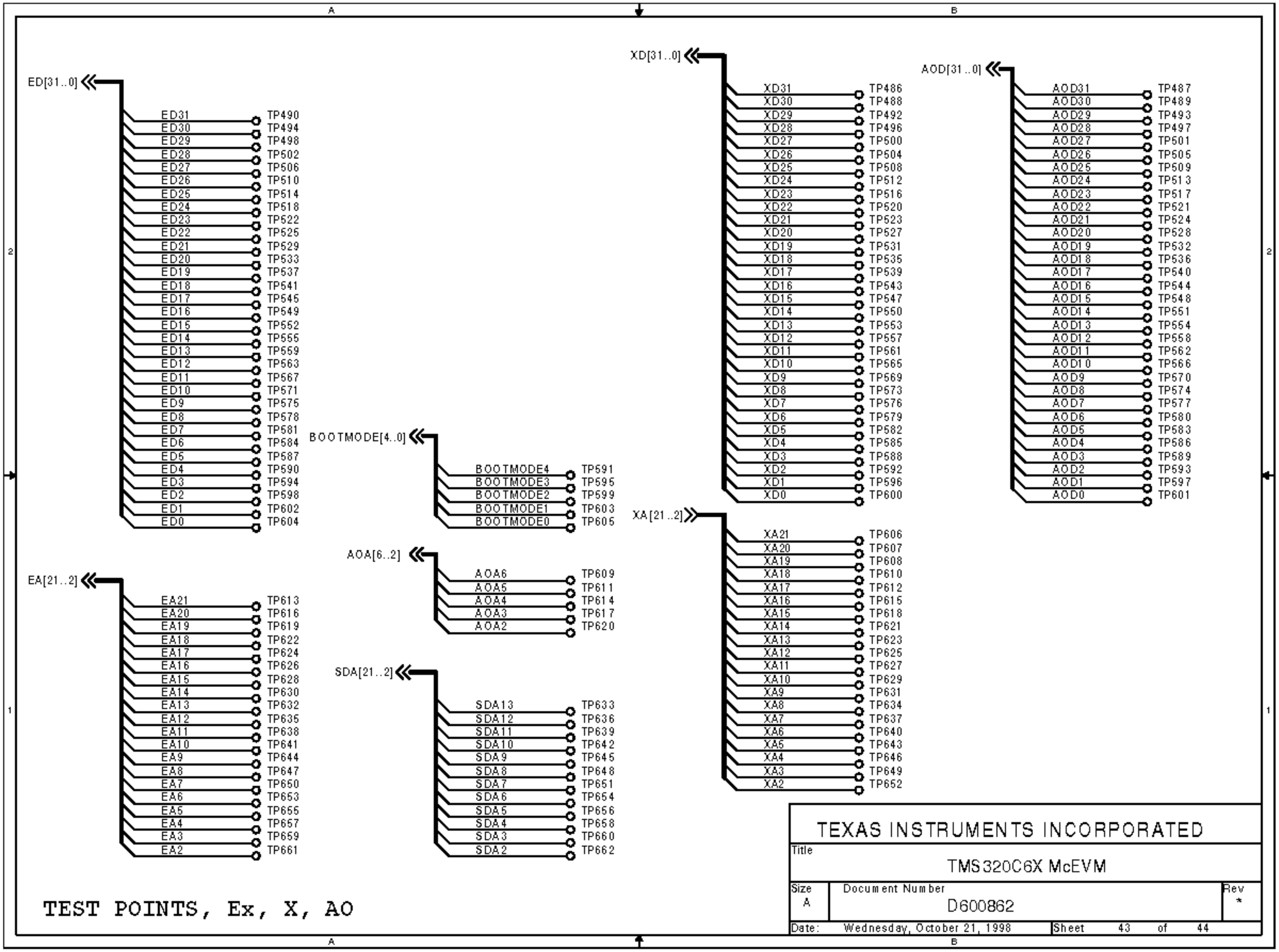
POWER-CAPS VCC2 & VCC3

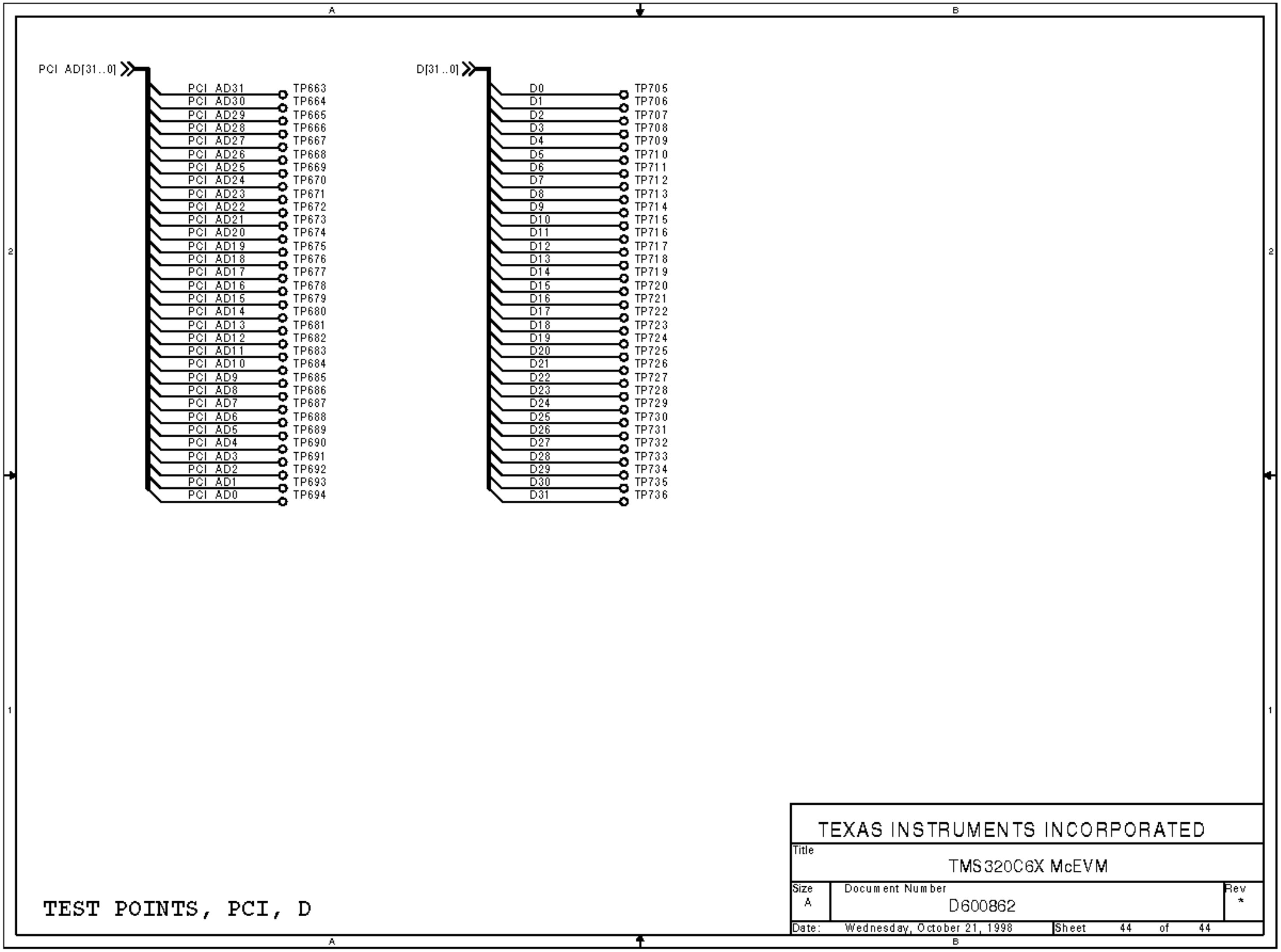
TEXAS INSTRUMENTS INCORPORATED

Title TMS320C6X McEVM		
Size A	Document Number D600862	Rev *
Date: Wednesday, October 21, 1998	Sheet 40 of 44	


A
B
2
1
VCC5
C549 .01 uF
C550 .01 uF
C574 .01 uF
C17 .01 uF
C541 .01 uF
C539 .01 uF
C522 .01 uF
C512 .01 uF
C7 .01 uF
C533 .01 uF
C527 .01 uF
C526 .01 uF
C569 .01 uF
C535 .01 uF
C520 .01 uF
C6 .01 uF
C9 .01 uF
C5 .01 uF
C543 .01 uF
C540 .01 uF
C529 .01 uF
C505 .01 uF
VCC5
C558 .1uF
C576 .1uF
C575 .1uF
C16 .1uF
C538 .1uF
C521 .1uF
C3 .1uF
C504 .1uF
C11 .1uF
C534 .1uF
C523 .1uF
C524 .1uF
C536 .1uF
C519 .1uF
C8 .1uF
C4 .1uF
C10 .1uF
C542 .1uF
C532 .1uF
C530 .1uF
C510 .1uF
C568 .1uF
MUST MOUNT NEAR VBAP
VCCA
C501 .01 uF
C516 .01 uF
C506 .1uF
C513 .1uF
VCC5
CT6 + 10uF
CT7 + 10uF
CT10 + 10uF
CT11 + 10uF
POWER-CAPS VCC5, VCCA
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 41 of 44


A
B
2
1
CLKIN TP272
CLKMODE TP276
DSP_RESET# TP280
NMI TP285
PCI_INT TP289
PCIMRD_INT TP293
PCIMWR_INT TP298
CE1_DIR TP303
LENDIAN TP308
TMS TP313
TDI TP318
TRST# TP328
PCI_SYSRST# TP333
USER_LED1# TP341
USER_LED0# TP346
OSC_A_EN# TP350
OSC_B_EN# TP353
ARDY TP362
DSP2AOD# TP366
DSP2XD# TP388
CE1# TP284
CE2# TP398
CE3# TP402
BE3# TP406
BE2# TP408
BE1# TP412
BE0# TP416
AWE# TP421
SDCLK TP425
SDRAS# TP429
SDCAS# TP434
SDWE# TP437
ARE# TP441
AOE# TP445
BRD_RST TP449
MAN_RST# TP456
CE2_SDEN TP466
DB_INT TP469
TINP0_B TP474
XFSR1 TP273
TOUT0 TP277
TOUT1 TP281
INUM3 TP290
INUM1 TP294
XRESET# TP299
XCNTL1 TP304
XSTAT1 TP309
XCE2# TP314
DMAC3 TP319
DMAC1 TP324
XCLKS0 TP329
XDX0 TP334
XDR0 TP337
XCLKS1 TP342
XDX1 TP347
XDR1 TP351
TINP0 TP354
TINP1 TP358
IACK TP363
INUM2 TP367
INUM0 TP372
DSP_PD TP377
XCNTL0 TP382
XCE3# TP389
DMAC2 TP392
DMAC0 TP396
XCLKOUT2 TP399
HINT# TP403
PC_INT TP409
PCI_FLT# TP413
MCBSP0_SEL TP417
CE3_SDEN TP422
PTRDY# TP426
PTADR# TP430
XWE# TP453
XRDY TP457
XCE1# TP460
XRE# TP462
XOE# TP464
XBE2# TP467
XBE0# TP470
XBE3# TP472
XBE1# TP475
XCLKX0 TP274
XFSX0 TP278
XCLKR0 TP282
XFSR0 TP287
XCLKX1 TP291
XFSX1 TP295
XCLKR1 TP300
TBC_EMU0 TP343
TBC_TDI TP348
TBC_EMU1 TP352
TBCA0_HBE0# TP359
TBCA1_HBE1# TP364
TBCA2_HHWIL TP368
TBCA3_HCNTL0 TP373
TBCA4_HCNTL1 TP378
TBC_CLK TP383
TBC_RST# TP386
TBC_RD# TP390
TBC_WR# TP393
TBC_TMS TP404
TBC_TCK TP410
TBC_TRST# TP414
TINP1_B TP418
TDO TP431
TCK TP435
JTAG_SEL TP438
EMU0 TP442
EMU1 TP446
INTA# TP450
SERR# TP454
PERR# TP458
GNT# TP461
REQ# TP463
DEVSEL# TP465
IDSEL TP468
LOCK# TP471
STOP# TP473
TRDY# TP476
IRDY# TP478
FRAME# TP479
RST# TP480
PCI_CLK TP482
PAR TP484
SSCLK TP271
CE0# TP275
SSOE# TP279
SSADS# TP283
SSWE# TP288
S_BOOTMODE4 TP292
S_BOOTMODE3 TP296
S_BOOTMODE2 TP301
S_BOOTMODE1 TP306
S_BOOTMODE0 TP311
S_CLKMODE TP316
S_LENDIAN TP321
S_JTAGSEL TP326
S_CLKSEL TP331
S_USER2 TP336
S_USER1 TP339
S_USER0 TP344
TBC_TDO TP355
PTBE3# TP379
PTBE2# TP384
PTBE1# TP387
PTBE0# TP391
PCI_DET# TP394
VCC2BAD# TP397
C/BE0# TP405
C/BE1# TP407
C/BE2# TP411
C/BE3# TP415
WRFIFO# TP419
RDFIFO# TP423
PCI_WR# TP427
PCI_RD# TP432
AOBE3# TP436
AOBE2# TP439
AOBE1# TP443
AOBE0# TP447
PCI_CS# TP451
G_BE3# TP312
G_BE2# TP317
G_BE1# TP322
G_BE0# TP327
XSTAT0 TP340
TBC_RDY# TP345
TBC_INT# TP349
HRDY# TP356
PCI_IRQ# TP361
PTATN# TP370
PTBURST# TP375
PTNUM1 TP380
PTNUM0 TP385
PTWR TP401
FMIC_CLK4 TP749
BPCLK TP440
WRFULL TP444
RDEMPTY TP448
HCS# TP452
HDS1# TP455
HRW TP459
TEST POINTS, MISC
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 42 of 44


ED[31..0]
ED31 TP490
ED30 TP494
ED29 TP498
ED28 TP502
ED27 TP506
ED26 TP510
ED25 TP514
ED24 TP518
ED23 TP522
ED22 TP525
ED21 TP529
ED20 TP533
ED19 TP537
ED18 TP541
ED17 TP545
ED16 TP549
ED15 TP552
ED14 TP555
ED13 TP559
ED12 TP563
ED11 TP567
ED10 TP571
ED9 TP575
ED8 TP578
ED7 TP581
ED6 TP584
ED5 TP587
ED4 TP590
ED3 TP594
ED2 TP598
ED1 TP602
ED0 TP604
EA[21..2]
EA21 TP613
EA20 TP616
EA19 TP619
EA18 TP622
EA17 TP624
EA16 TP626
EA15 TP628
EA14 TP630
EA13 TP632
EA12 TP635
EA11 TP638
EA10 TP641
EA9 TP644
EA8 TP647
EA7 TP650
EA6 TP653
EA5 TP655
EA4 TP657
EA3 TP659
EA2 TP661
BOOTMODE[4..0]
BOOTMODE4 TP591
BOOTMODE3 TP595
BOOTMODE2 TP599
BOOTMODE1 TP603
BOOTMODE0 TP605
AOA[6..2]
AOA6 TP609
AOA5 TP611
AOA4 TP614
AOA3 TP617
AOA2 TP620
SDA[21..2]
SDA13 TP633
SDA12 TP636
SDA11 TP639
SDA10 TP642
SDA9 TP645
SDA8 TP648
SDA7 TP651
SDA6 TP654
SDA5 TP656
SDA4 TP658
SDA3 TP660
SDA2 TP662
XD[31..0]
XD31 TP486
XD30 TP488
XD29 TP492
XD28 TP496
XD27 TP500
XD26 TP504
XD25 TP508
XD24 TP512
XD23 TP516
XD22 TP520
XD21 TP523
XD20 TP527
XD19 TP531
XD18 TP535
XD17 TP539
XD16 TP543
XD15 TP547
XD14 TP550
XD13 TP553
XD12 TP557
XD11 TP561
XD10 TP565
XD9 TP569
XD8 TP573
XD7 TP576
XD6 TP579
XD5 TP582
XD4 TP585
XD3 TP588
XD2 TP592
XD1 TP596
XD0 TP600
XA[21..2]
XA21 TP606
XA20 TP607
XA19 TP608
XA18 TP610
XA17 TP612
XA16 TP615
XA15 TP618
XA14 TP621
XA13 TP623
XA12 TP625
XA11 TP627
XA10 TP629
XA9 TP631
XA8 TP634
XA7 TP637
XA6 TP640
XA5 TP643
XA4 TP646
XA3 TP649
XA2 TP652
AOD[31..0]
AOD31 TP487
AOD30 TP489
AOD29 TP493
AOD28 TP497
AOD27 TP501
AOD26 TP505
AOD25 TP509
AOD24 TP513
AOD23 TP517
AOD22 TP521
AOD21 TP524
AOD20 TP528
AOD19 TP532
AOD18 TP536
AOD17 TP540
AOD16 TP544
AOD15 TP548
AOD14 TP551
AOD13 TP554
AOD12 TP558
AOD11 TP562
AOD10 TP566
AOD9 TP570
AOD8 TP574
AOD7 TP577
AOD6 TP580
AOD5 TP583
AOD4 TP586
AOD3 TP589
AOD2 TP593
AOD1 TP597
AOD0 TP601
TEST POINTS, Ex, X, AO
TEXAS INSTRUMENTS INCORPORATED
Title
TMS320C6X McEVM
Size
A
Document Number
D600862
Rev
*
Date: Wednesday, October 21, 1998
Sheet 43 of 44



TEST POINTS, PCI, D

TEXAS INSTRUMENTS INCORPORATED

Title
TMS320C6X McEVM

Size A
Document Number D600862
Rev *

Date: Wednesday, October 21, 1998
Sheet 44 of 44

Appendix C

TMS320C62x McEVM CPLD Equations

This appendix provides the complex programmable logic device (CPLD) equations. The CPLD, which is the only programmable logic device on the ’C62x McEVM, is designated as U12 on the board.

Topic **Page**

C.1 Overview of the McEVM CPLD

The CPLD is an Altera EPM7256SQC208–10 device that has a 208-pin PQFP package. A summary of the pin allocation is shown in Table C–1.

Table C–1. TMS320C62x McEVM CPLD Pin Summary

Pin Type	Number of Pins
Inputs	64
Bidirectional	16
Outputs	60
Unused	20
JTAG	4
5 V	4
3.3 V	10
GND	14
No connects	16
Total pins	**208**

The EPM7256SQC208–10 CPLD provides 5000 usable gates, of which approximately 75% are used in this design. The device contains 256 macrocells arranged in 16 logic array blocks. The maximum pin-to-pin delay is 10.5 ns. The CPLD uses 5 V for internal operation and 3.3 V for its I/O buffers. It can interface to both 3.3- and 5-V devices.

The CPLD was designed using Synario™ ABEL™ version 6.5 and Altera MAX+ PLUS™ II version 8.1. Synario was used to enter the design in the ABEL hardware design language, and MAX+ PLUS II was used for design compilation, simulation, and programming file generation.

The CPLD's VHD source files are modular with a top-level module and six low-level modules as shown in Figure C–1. These seven source files are provided in section C.2, *McEVM CPLD Equations*.

Figure C–1. TMS320C62x McEVM CPLD Source Files

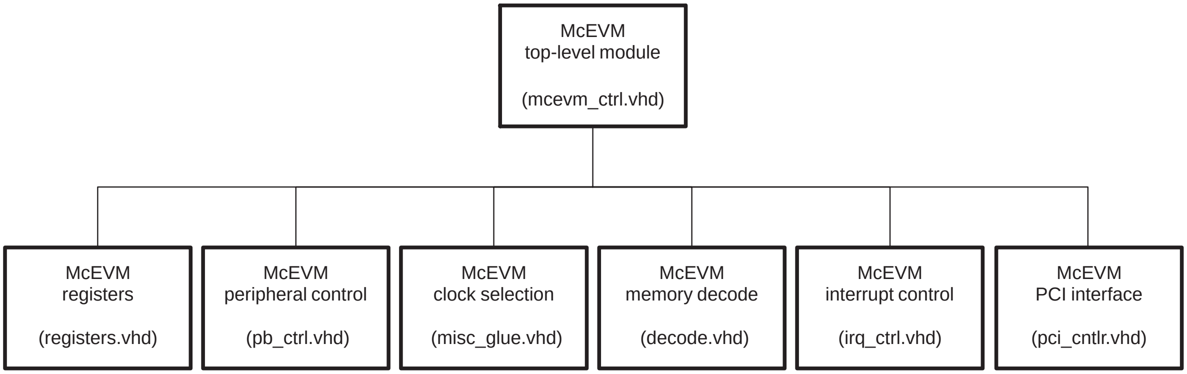


Table C–2 provides the CPLD pin definitions in numerical pin order. Table C–3 provides the CPLD pin definitions in sequential alphabetical order.

Table C–2. TMS320C62x CPLD I/O Pins Sorted by Signal Name

Name	Type	Number
alaw_en	O	44
ao_adr2	O	109
ao_adr3	O	112
ao_adr4	O	111
ao_adr5	O	113
ao_adr6	O	108
ao_be0_l	O	169
ao_be1_l	O	168
ao_be2_l	O	100
ao_be3_l	O	97
ao_rd_l	O	198
ao_sel_l	O	202
ao_wr_l	O	199
ardy	O	64
are_l	I	40
awe_l	I	184
be0_l	I	128
be1_l	I	138
be2_l	I	76
be3_l	I	204
bootmode0	O	29
bootmode1	O	22
bootmode2	O	37
bootmode3	O	39
bootmode4	O	9
bpclk	I	181
brd_rst_l	I	182
ce1_l	I	162
ce2_l	I	45
ce2_sden	O	70
ce3_l	I	84
ce3_sden	O	121

clk_a	I	166
clkmode	O	20
db_falc_int	O	130
db_int	I	150
dq0	I/O	25
dq1	I/O	21
dq2	I/O	38
dq3	I/O	48
dq4	I/O	7
dq5	I/O	15
dq6	I/O	19
dq7	I/O	47
dsp_hint_l	I	137
dsp_hrdy_l	I	129
dsp_pd	I	118
dsp_rst_l	O	27
dsp2aod_l	O	61
dsp2gd_l	O	60
dsp2xd_l	O	62
ea16	I	65
ea17	I	161
ea18	I	167
ea19	I	123
ea2	I	90
ea20	I	149
ea21	I	49
ea3	I	177
ea4	I	13
ea5	I	24
ea6	I	91
ed0	I/O	79
ed1	I/O	197
ed2	I/O	119

ed3	I/O	124
ed4	I/O	120
ed5	I/O	12
ed6	I/O	192
ed7	I/O	188
ext_rst_l	O	71
falc_a0	O	173
falc_bxe_l	O	178
falc_cs_l	O	171
falc_int	I	205
falc_moto	O	16
falc_rst	O	86
falc_sync	O	145
float_l	I	183
fmic_2m_clk	I	55
fmic_cs_l	O	170
fmic_err	I	17
fmic_frm_en_l	O	146
fmic_rdy	I	67
fmic_rst_l	O	66
hcs_l	O	144
hds1_l	O	159
hrw	O	98
jtagsel	O	42
ld20_swap	O	88
ld31_swap	O	122
led0_l	O	87
led1_l	O	77
lendian	O	11
loop_led_l	O	187
man_rst_l	O	114
nmi	O	6
osc_a_en_l	O	133

osc_b_en_l	O	132
pb_rd_l	O	93
pb_wr_l	O	96
pc_int	O	172
pci_det_l	I	18
pci_flt_l	O	33
pci_int	O	69
pci_irq_l	I	135
pci_rst_l	I	102
pcimrd_int	O	59
pcimwr_int	O	117
pt_adr_l	O	4
pt_rdy_l	O	95
ptatn_l	I	43
ptbe0_l	I	99
ptbe1_l	I	81
ptbe2_l	I	57
ptbe3_l	I	163
ptburst_l	I	31
ptnum0	I	193
ptnum1	I	175
ptwr	I	28
ralm_led_l	O	78
rdempty	I	68
rdfifo_l	O	3
s_bootmode0	I	115
s_bootmode1	I	56
s_bootmode2	I	147
s_bootmode3	I	34
s_bootmode4	I	139
s_clkmode	I	136
s_clksel	I	195
s_endian	I	10

s_jtagsel	I	206
s_user0	I	73
s_user1	I	36
s_user2	I	151
sp0sel	O	80
sw_rst_l	I	140
sync_led_l	O	194
tbc_int_l	I	153
tbc_rd_l	O	201
tbc_rdy_l	I	101
tbc_rst_l	O	26
tbc_wr_l	O	92
tbca0_hbe0_l	O	160
tbca1_hbe1_l	O	154
tbca2_hhwil	O	141
tbca3_hcntl0	O	142
tbca4_hcntl1	O	110
ulaw_en	O	46
vcc2bad_l	I	164
wrfifo_l	O	203
wrfull	I	131
xcntl0	O	190
xcntl1	O	196
xrdy	I	8
xstat0	I	35
xstat1	I	58
yalm_led_l	O	89

Table C–3. TMS320C62x CPLD I/O Pins Sorted by Pin Number

Name	Type	Number
rdfifo_l	O	3
pt_adr_l	O	4
nmi	O	6
dq4	I/O	7
xrdy	I	8
bootmode4	O	9
s_endian	I	10
lendian	O	11
ed5	I/O	12
ea4	I	13
dq5	I/O	15
falc_moto	O	16
fmic_err	I	17
pci_det_l	I	18
dq6	I/O	19
clkmode	O	20
dq1	I/O	21
bootmode1	O	22
ea5	I	24
dq0	I/O	25
tbc_rst_l	O	26
dsp_rst_l	O	27
ptwr	I	28
bootmode0	O	29
ptburst_l	I	31
pci_flt_l	O	33

s_bootmode3	I	34
xstat0	I	35
s_user1	I	36
bootmode2	O	37
dq2	I/O	38
bootmode3	O	39
are_l	I	40
jtagsel	O	42
ptatn_l	I	43
alaw_en	O	44
ce2_l	I	45
ulaw_en	O	46
dq7	I/O	47
dq3	I/O	48
ea21	I	49
fmic_2m_clk	I	55
s_bootmode1	I	56
ptbe2_l	I	57
xstat1	I	58
pcimrd_int	O	59
dsp2gd_l	O	60
dsp2aod_l	O	61
dsp2xd_l	O	62
ardy	O	64
ea16	I	65
fmic_rst_l	O	66
fmic_rdy	I	67
rdempty	I	68

pci_int	O	69
ce2_sden	O	70
ext_rst_l	O	71
s_user0	I	73
be2_l	I	76
led1_l	O	77
ralm_led_l	O	78
ed0	I/O	79
sp0sel	O	80
ptbe1_l	I	81
ce3_l	I	84
falc_rst	O	86
led0_l	O	87
ld20_swap	O	88
yalm_led_l	O	89
ea2	I	90
ea6	I	91
tbc_wr_l	O	92
pb_rd_l	O	93
pt_rdy_l	O	95
pb_wr_l	O	96
ao_be3_l	O	97
hrw	O	98
ptbe0_l	I	99
ao_be2_l	O	100
tbc_rdy_l	I	101
pci_rst_l	I	102
ao_adr6	O	108

ao_adr2	O	109
tbca4_hcntl1	O	110
ao_adr4	O	111
ao_adr3	O	112
ao_adr5	O	113
man_rst_l	O	114
s_bootmode0	I	115
pcimwr_int	O	117
dsp_pd	I	118
ed2	I/O	119
ed4	I/O	120
ce3_sden	O	121
ld31_swap	O	122
ea19	I	123
ed3	I/O	124
be0_l	I	128
dsp_hrdy_l	I	129
db_falc_int	O	130
wrfull	I	131
osc_b_en_l	O	132
osc_a_en_l	O	133
pci_irq_l	I	135
s_clkmode	I	136
dsp_hint_l	I	137
be1_l	I	138
s_bootmode4	I	139
sw_rst_l	I	140
tbca2_hhwil	O	141

tbca3_hcntl0	O	142
hcs_l	O	144
falc_sync	O	145
fmic_frm_en_l	O	146
s_bootmode2	I	147
ea20	I	149
db_int	I	150
s_user2	I	151
tbc_int_l	I	153
tbca1_hbe1_l	O	154
hds1_l	O	159
tbca0_hbe0_l	O	160
ea17	I	161
ce1_l	I	162
ptbe3_l	I	163
vcc2bad_l	I	164
clk_a	I	166
ea18	I	167
ao_be1_l	O	168
ao_be0_l	O	169
fmic_cs_l	O	170
falc_cs_l	O	171
pc_int	O	172
falc_a0	O	173
ptnum1	I	175
ea3	I	177
falc_bxe_l	O	178
bpclk	I	181

brd_rst_l	I	182
float_l	I	183
awe_l	I	184
loop_led_l	O	187
ed7	I/O	188
xcntl0	O	190
ed6	I/O	192
ptnum0	I	193
sync_led_l	O	194
s_clksel	I	195
xcntl1	O	196
ed1	I/O	197
ao_rd_l	O	198
ao_wr_l	O	199
tbc_rd_l	O	201
ao_sel_l	O	202
wrfifo_l	O	203
be3_l	I	204
falc_int	I	205
s_jtagsel	I	206

C.2 McEVM CPLD Equations

The following listing provides a table of contents for the CPLD source files included in this section.

```
-------------------------------------------------------------------------
--                                     Design For:
--                         Texas Instruments Incorporated
--
--                                     Design By:
--               DNA Enterprises Inc
--               269 West Renner Parkway
--               Richardson, TX 75080
--
--   File name     :  mcevm_ctrl.vhd
--   Title         :  Top Level McEVM Control CPLD design
--   Module        :
--   Description   :  This is the top–level module for the TMS320C6x
--        McEVM’s CPLD. This top level only contains
--        instantiations of the lower level components
--        and the concurrent statements to tri–state
--        outputs when required.
--
-------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     03/30/98
-- Author:       Don Curry (DNA)
-- Description:  Initial conversion from ABEL version of EVM CPLD.
--
-- Revision: 1
-- Date:     05/05/98
-- Author:       Don Curry (DNA)
-- Description: Added McEVM specific signals/components
--
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY mcevm_ctrl IS
   PORT(
-- active low signals are indicated with a trailing underscore.
   -----------------------------------------------------------------------
   -- INPUTS
   -----------------------------------------------------------------------
   -- CPLD control
   float_l       : IN STD_LOGIC ; -- floats all outputs
   -- Resets
   brd_rst_l : IN STD_LOGIC ; -- reset from voltage super.
   sw_rst_l  : IN STD_LOGIC ; -- reset from pushbutton switch
   vcc2bad_l : IN STD_LOGIC ; -- VCC bad from voltage super.
   pci_rst_l : IN STD_LOGIC ; -- PCI reset from PCI cntrlr
   -- Clocks
   clk_a     : IN STD_LOGIC ; -- DSP clk source (OSC A)
   -- EMIF Control
   ce1_l     : IN STD_LOGIC ; -- EMIF CE1 memory space enable
   ce2_l     : IN STD_LOGIC ; -- EMIF CE2 memory space enable
   ce3_l     : IN STD_LOGIC ; -- EMIF CE3 memory space enable
   are_l     : IN STD_LOGIC ; -- EMIF async read strobe
   awe_l     : IN STD_LOGIC ; -- EMIF async write strobe
   be0_l     : IN STD_LOGIC ; -- EMIF byte 0 enable
```

```
be1_l     : IN STD_LOGIC ; -- EMIF byte 1 enable
be2_l     : IN STD_LOGIC ; -- EMIF byte 2 enable
be3_l     : IN STD_LOGIC ; -- EMIF byte 3 enable
-- EMIF Address
ea21      : IN STD_LOGIC ;
ea20      : IN STD_LOGIC ;
ea19      : IN STD_LOGIC ;
ea18      : IN STD_LOGIC ;
ea17      : IN STD_LOGIC ;
ea16      : IN STD_LOGIC ;
ea6    : IN STD_LOGIC ;
ea5    : IN STD_LOGIC ;
ea4    : IN STD_LOGIC ;
ea3    : IN STD_LOGIC ;
ea2    : IN STD_LOGIC ;
-- DSP Host port control
dsp_hrdy_l   : IN STD_LOGIC ; -- DSP HPI ready
dsp_hint_l   : IN STD_LOGIC ; -- DSP HPI host interrupt
dsp_pd    : IN STD_LOGIC ; -- DSP power down indicator
-- From S5933 PCI Controller
bpclk     : IN STD_LOGIC ; -- Buffered PCI clock
ptatn_l      : IN STD_LOGIC ; -- Pass-thru attention
ptburst_l : IN STD_LOGIC ; -- Pass-thru burst
ptnum1    : IN STD_LOGIC ; -- Pass-thru region #
ptnum0    : IN STD_LOGIC ; -- Pass-thru region #
ptwr      : IN STD_LOGIC ; -- Pass-thru access type
                 -- (R=0, W=1)
ptbe3_l      : IN STD_LOGIC ; -- Pass-thru byte enable 3
ptbe2_l      : IN STD_LOGIC ; -- Pass-thru byte enable 2
```

```
ptbe1_l       : IN STD_LOGIC ; -- Pass-thru byte enable 1
ptbe0_l       : IN STD_LOGIC ; -- Pass-thru byte enable 0
pci_irq_l : IN STD_LOGIC ; -- Interrupt to add-on device
pci_det_l : IN STD_LOGIC ; -- PCI Bus detection
                 -- (0=PCI, 1=standalone)
rdempty       : IN STD_LOGIC ; -- Read FIFO empty flag
wrfull    : IN STD_LOGIC ; -- Write FIFO full flag
-- From JTAG TBC
tbc_rdy_l : IN STD_LOGIC ; -- TBC ready
tbc_int_l : IN STD_LOGIC ; -- TBC interrupt
-- From DIP switch
s_bootmode4   : IN STD_LOGIC ; -- Bootmode 4 (SW2-1)
s_bootmode3   : IN STD_LOGIC ; -- Bootmode 3 (SW2-2)
s_bootmode2   : IN STD_LOGIC ; -- Bootmode 2 (SW2-3)
s_bootmode1   : IN STD_LOGIC ; -- Bootmode 1 (SW2-4)
s_bootmode0   : IN STD_LOGIC ; -- Bootmode 0 (SW2-5)
s_clkmode : IN STD_LOGIC ; -- Clock mode (SW2-6)
                 -- x1 (no PLL) or x4 (PLL)
s_clksel  : IN STD_LOGIC ; -- Clock select (SW2-7)
                 -- Osc A or Osc B
s_endian  : IN STD_LOGIC ; -- Memory access select (SW2-8)
                 -- little or big endian
s_jtagsel : IN STD_LOGIC ; -- JTAG select (SW2-9)
                 -- Internal or external emul.
s_user2       : IN STD_LOGIC ; -- User-defined switch (SW2-10)
s_user1       : IN STD_LOGIC ; -- User-defined switch (SW2-11)
s_user0       : IN STD_LOGIC ; -- User-defined switch (SW2-12)
-- From T1/E1 transceiver
falc_int  : IN STD_LOGIC ; -- T1/E1 level interrupt (active high)
```

```
-- From FMIC
fmic_rdy  : IN STD_LOGIC ; -- FMIC ready signal
fmic_err  : IN STD_LOGIC ; -- FMIC error signal
fmic_2m_clk  : IN STD_LOGIC ; -- FMIC 2 MHz clock

-- From daughterboard expansion
xstat1    : IN STD_LOGIC ; -- External status 1
xstat0    : IN STD_LOGIC ; -- External status 0
db_int    : IN STD_LOGIC ; -- Daughterboard IRQ (EXT_INT7)
xrdy      : IN STD_LOGIC ; -- External async mem. acc. rdy.

------------------------------------------------------------------------
-- BI-DIRECTIONALS
------------------------------------------------------------------------
-- EMIF data bus (From DSP)
ed7    : INOUT STD_LOGIC ;
ed6    : INOUT STD_LOGIC ;
ed5    : INOUT STD_LOGIC ;
ed4    : INOUT STD_LOGIC ;
ed3    : INOUT STD_LOGIC ;
ed2    : INOUT STD_LOGIC ;
ed1    : INOUT STD_LOGIC ;
ed0    : INOUT STD_LOGIC ;
-- Add-on data bus (From PCI controller)
dq7    : INOUT STD_LOGIC ;
dq6    : INOUT STD_LOGIC ;
dq5    : INOUT STD_LOGIC ;
dq4    : INOUT STD_LOGIC ;
dq3    : INOUT STD_LOGIC ;
```

```
dq2    : INOUT STD_LOGIC ;
dq1    : INOUT STD_LOGIC ;
dq0    : INOUT STD_LOGIC ;
-----------------------------------------------------------------------
-- OUTPUTS
-----------------------------------------------------------------------
-- McEVM specific control outputs
ld31_swap : OUT STD_LOGIC ;   -- FMIC stream 3/1 mux ctrl
ld20_swap : OUT STD_LOGIC ;   -- FMIC stream 2/0 mux ctrl
loop_led_l   : OUT STD_LOGIC ;   -- drive loop led
sync_led_l   : OUT STD_LOGIC ;   -- drive sync led
ralm_led_l   : OUT STD_LOGIC ;   -- drive red alarm led
yalm_led_l   : OUT STD_LOGIC ;   -- drive yellow alarm led
alaw_en      : OUT STD_LOGIC ;   -- A–law select
ulaw_en      : OUT STD_LOGIC ;   -- u–law select
-- To Data buffers/transceivers
dsp2aod_l : OUT STD_LOGIC ;   -- Connects DSP to Add–on bus
dsp2gd_l  : OUT STD_LOGIC ;   -- Connects DSP to global bus
dsp2xd_l  : OUT STD_LOGIC ;   -- Connects DSP to external bus
-- To JTAG TBC and DSP HPI
tbca0_hbe0_l : OUT STD_LOGIC ;   -- TBC addr 0/HPI byte 0 enable
tbca1_hbe1_l : OUT STD_LOGIC ;   -- TBC addr 1/HPI byte 1 enable
tbca2_hhwil  : OUT STD_LOGIC ;   -- TBC addr 2/HPI half–word sel.
tbca3_hcntl0 : OUT STD_LOGIC ;   -- TBC addr 3/HPI control 0
tbca4_hcntl1 : OUT STD_LOGIC ;   -- TBC addr 4/HPI control 1
-- To JTAG TBC
tbc_wr_l  : OUT STD_LOGIC ;   -- TBC write strobe
tbc_rd_l  : OUT STD_LOGIC ;   -- TBC read strobe
tbc_rst_l : OUT STD_LOGIC ;   -- TBC hardware reset
```

```
-- To HPI
hcs_l     : OUT STD_LOGIC ;   -- HPI chip select
hds1_l    : OUT STD_LOGIC ;   -- HPI data strobe 1
hrw    : OUT STD_LOGIC ;   -- HPI read (1)/write (0)
-- To S5933 PCI controller
pci_flt_l : OUT STD_LOGIC ;   -- Float PCI controller outputs
pt_adr_l  : OUT STD_LOGIC ;   -- Pass-thru address request
pt_rdy_l  : OUT STD_LOGIC ;   -- Pass-thru ready indication
ao_sel_l  : OUT STD_LOGIC ;   -- Add-on sel. for reg. access
ao_wr_l       : OUT STD_LOGIC ;   -- Add-on write strobe
ao_rd_l       : OUT STD_LOGIC ;   -- Add-on read strobe
ao_adr6       : OUT STD_LOGIC ;   -- Add-on Address bus
ao_adr5       : OUT STD_LOGIC ;   -- Add-on Address bus
ao_adr4       : OUT STD_LOGIC ;   -- Add-on Address bus
ao_adr3       : OUT STD_LOGIC ;   -- Add-on Address bus
ao_adr2       : OUT STD_LOGIC ;   -- Add-on Address bus
ao_be3_l  : OUT STD_LOGIC ;   -- Add-on byte enable 3
ao_be2_l  : OUT STD_LOGIC ;   -- Add-on byte enable 2
ao_be1_l  : OUT STD_LOGIC ;   -- Add-on byte enable 1
ao_be0_l  : OUT STD_LOGIC ;   -- Add-on byte enable 0
rdfifo_l  : OUT STD_LOGIC ;   -- Read FIFO strobe
wrfifo_l  : OUT STD_LOGIC ;   -- Write FIFO strobe
pc_int    : OUT STD_LOGIC ;   -- Add-on to PCI interrupt
-- To DSP
-- Bootmode for No-boot, HPI boot or ROM boot
bootmode4 : OUT STD_LOGIC ;
bootmode3 : OUT STD_LOGIC ;
bootmode2 : OUT STD_LOGIC ;
bootmode1 : OUT STD_LOGIC ;
```

```
bootmode0 : OUT STD_LOGIC ;
clkmode       : OUT STD_LOGIC ;   -- Clock mode
lendian       : OUT STD_LOGIC ;   -- Little endian selection
nmi    : OUT STD_LOGIC ;   -- NMI interrupt from host
db_falc_int   : OUT STD_LOGIC ;   -- FALC/DB (EXT_INT7)
pci_int       : OUT STD_LOGIC ;   -- Interrupt from PCI cntrlr
                 -- (EXT_INT4)
pcimrd_int    : OUT STD_LOGIC ;   -- PCI master read interrupt
                 -- (EXT_INT5)
pcimwr_int    : OUT STD_LOGIC ;   -- PCI master write interrupt
                 -- (EXT_INT6)
dsp_rst_l : OUT STD_LOGIC ;   -- DSP reset
ardy      : OUT STD_LOGIC ;   -- EMIF async memory access ready
-- To multiplexers
osc_a_en_l    : OUT STD_LOGIC ;   -- DSP oscillator A enable
osc_b_en_l    : OUT STD_LOGIC ;   -- DSP oscillator B enable
jtagsel       : OUT STD_LOGIC ;   -- JTAG selection (int/ext)
sp0sel    : OUT STD_LOGIC ;   -- McBSP0 selection
-- To FMIC/FALC timing
fmic_frm_en_l: OUT STD_LOGIC ;   -- Enable FMIC frame to D.B.
falc_sync : OUT STD_LOGIC ;   -- FALC sync mux output
-- To LED’s
led1_l    : OUT STD_LOGIC ;   -- User LED 1 control
led0_l    : OUT STD_LOGIC ;   -- User LED 0 control
-- To Peripherial bus components (T1/E1 xcvr & FMIC)
fmic_rst_l    : OUT STD_LOGIC ;   -- FMIC reset
falc_rst  : OUT STD_LOGIC ;   -- FALC reset
fmic_cs_l : OUT STD_LOGIC ;   -- Chip select
falc_cs_l : OUT STD_LOGIC ;   -- Chip select
```

```
   falc_bxe_l    : OUT STD_LOGIC ;   -- FALC BHE/BLE
   falc_a0       : OUT STD_LOGIC ;   -- FALC address 0
   falc_moto : OUT STD_LOGIC ;   -- FALC interface mode
   pb_wr_l       : OUT STD_LOGIC ;   -- Write strobe
   pb_rd_l       : OUT STD_LOGIC ;   -- Read strobe

   -- To daughterboard
   xcntl1    : OUT STD_LOGIC ;   -- External control 1
   xcntl0    : OUT STD_LOGIC ;   -- External control 0
   ext_rst_l : OUT STD_LOGIC ;   -- External reset
   -- To SDRAMs
   ce3_sden  : OUT STD_LOGIC ;   -- Bank 1 enable (CKE)
   ce2_sden  : OUT STD_LOGIC ;   -- Bank 0 enable (CKE)
   -- To voltage supervisor
   man_rst_l : OUT STD_LOGIC  -- Pushbutton or PCI reset
   );
END mcevm_ctrl ;
ARCHITECTURE struct OF mcevm_ctrl IS
-- Define types used
-- Define constants for readability
  CONSTANT true_h   : STD_LOGIC := ’1’ ;
  CONSTANT false_h  : STD_LOGIC := ’0’ ;
  CONSTANT true_l   : STD_LOGIC := ’0’ ;
  CONSTANT false_l  : STD_LOGIC := ’1’ ;
-- Internal signal declarations
  SIGNAL ce2sden : STD_LOGIC ;
  SIGNAL ce3sden : STD_LOGIC ;
  SIGNAL ea_hi      : STD_LOGIC_VECTOR(5 DOWNTO 0) ;
  SIGNAL ea_lo      : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
SIGNAL ao_bsy      : STD_LOGIC ;
SIGNAL emif_ack    : STD_LOGIC ;
SIGNAL are_pci_l   : STD_LOGIC ;
SIGNAL awe_pci_l   : STD_LOGIC ;
SIGNAL are_osca_l  : STD_LOGIC ;
SIGNAL awe_osca_l  : STD_LOGIC ;
SIGNAL emif_req    : STD_LOGIC ;
SIGNAL cpld_cs : STD_LOGIC ;
SIGNAL pciclk      : STD_LOGIC ;
SIGNAL hinten      : STD_LOGIC ;
SIGNAL tbcinten    : STD_LOGIC ;
SIGNAL dspnmi      : STD_LOGIC ;
SIGNAL nmisel      : STD_LOGIC ;
SIGNAL nmien       : STD_LOGIC ;
SIGNAL pcimren : STD_LOGIC ;
SIGNAL pcimwen : STD_LOGIC ;
SIGNAL read_fifo_l    : STD_LOGIC ;
SIGNAL write_fifo_l   : STD_LOGIC ;
SIGNAL pci_mrd_int    : STD_LOGIC ;
SIGNAL pci_mwr_int    : STD_LOGIC ;
SIGNAL switch      : STD_LOGIC_VECTOR(12 DOWNTO 1) ;
SIGNAL dip_switch  : STD_LOGIC_VECTOR(12 DOWNTO 1) ;
SIGNAL soft_switch    : STD_LOGIC_VECTOR(12 DOWNTO 1) ;
SIGNAL swsel_dip_l    : STD_LOGIC ;
SIGNAL ptadr_l : STD_LOGIC ;
SIGNAL ptnum       : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
SIGNAL ptbe_l      : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
SIGNAL pt_be_l : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
SIGNAL be_l    : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
SIGNAL ao_be_l : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
SIGNAL ao_adr     : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
SIGNAL pt_addr : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
SIGNAL tbca_hpic  : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
SIGNAL xstat      : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
SIGNAL xcntl      : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
SIGNAL led     : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
SIGNAL pcireg_ce  : STD_LOGIC ;
SIGNAL pcireg_oe  : STD_LOGIC ;
SIGNAL ed_oe      : STD_LOGIC ;
SIGNAL ed      : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL ed_out     : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL dq      : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL dq_out     : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL xreset     : STD_LOGIC ;
SIGNAL dsprst     : STD_LOGIC ;
SIGNAL tbcrst     : STD_LOGIC ;
SIGNAL aod_l      : STD_LOGIC ;
SIGNAL gd_l    : STD_LOGIC ;
SIGNAL xbd_l      : STD_LOGIC ;
SIGNAL tbcwr_l : STD_LOGIC ;
SIGNAL tbcrd_l : STD_LOGIC ;
SIGNAL tbcrst_l   : STD_LOGIC ;
SIGNAL hpics_l : STD_LOGIC ;
SIGNAL hpids_l : STD_LOGIC ;
SIGNAL hpirw      : STD_LOGIC ;
SIGNAL pciflt_l   : STD_LOGIC ;
SIGNAL ptrdy_l : STD_LOGIC ;
SIGNAL aosel_l : STD_LOGIC ;
SIGNAL aowr_l     : STD_LOGIC ;
SIGNAL aord_l     : STD_LOGIC ;
SIGNAL ptrd_l     : STD_LOGIC ;
SIGNAL pcint      : STD_LOGIC ;
SIGNAL nmint      : STD_LOGIC ;
SIGNAL pciint     : STD_LOGIC ;
SIGNAL dsprst_l   : STD_LOGIC ;
SIGNAL async_rdy  : STD_LOGIC ;
SIGNAL osca_en_l  : STD_LOGIC ;
SIGNAL oscb_en_l  : STD_LOGIC ;
SIGNAL sp0_sel : STD_LOGIC ;
SIGNAL xrst_l     : STD_LOGIC ;
SIGNAL manrst_l   : STD_LOGIC ;
SIGNAL fmiccs_l   : STD_LOGIC ;
SIGNAL falccs_l   : STD_LOGIC ;
SIGNAL pbwr_l     : STD_LOGIC ;
SIGNAL pbrd_l     : STD_LOGIC ;
SIGNAL falc_req   : STD_LOGIC ;
SIGNAL falcbxe_l  : STD_LOGIC ;
SIGNAL falca0     : STD_LOGIC ;
SIGNAL fmic_req   : STD_LOGIC ;
SIGNAL pb_ack     : STD_LOGIC ;
SIGNAL mstr_sel   : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
SIGNAL ld31swap   : STD_LOGIC ;
SIGNAL ld20swap   : STD_LOGIC ;
SIGNAL fmicrst_l  : STD_LOGIC ;
SIGNAL falcrst : STD_LOGIC ;
SIGNAL fmicrst_b  : STD_LOGIC ;
SIGNAL falcrst_b  : STD_LOGIC ;
```

```
  SIGNAL ltchd_db_int   : STD_LOGIC ;
  SIGNAL ltchd_falc_int: STD_LOGIC ;
  SIGNAL clr_dbint  : STD_LOGIC ;
  SIGNAL clr_falcint    : STD_LOGIC ;
  SIGNAL loopled_l  : STD_LOGIC ;
  SIGNAL syncled_l  : STD_LOGIC ;
  SIGNAL ralmled_l  : STD_LOGIC ;
  SIGNAL yalmled_l  : STD_LOGIC ;
  SIGNAL a_law_en   : STD_LOGIC ;
  SIGNAL u_law_en   : STD_LOGIC ;
  SIGNAL fmic_frmen_l   : STD_LOGIC ;
  SIGNAL falcsync   : STD_LOGIC ;
  SIGNAL dbfalc_int : STD_LOGIC ;
-------------------------------------------------------------------------
-- Component declarations
-------------------------------------------------------------------------
COMPONENT decode
   PORT(

-- active low signals are indicated with ’_l’ appended to signal name.

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------

        brd_rst_l       : IN STD_LOGIC ;        -- Reset from volt. super.
        pci_det_l       : IN STD_LOGIC ;        -- PCI detection indicator
        pciclk          : IN STD_LOGIC ;        -- Buffered PCI clk (33MHz max)
   clk_a      : IN STD_LOGIC ; -- Osc. A (33.25MHz)
```

```
    ce1_l            : IN STD_LOGIC ;        -- EMIF CE1 memory space enable
    ce2_l            : IN STD_LOGIC ;        -- EMIF CE2 memory space enable
    ce3_l            : IN STD_LOGIC ;        -- EMIF CE3 memory space enable
    are_l            : IN STD_LOGIC ;        -- EMIF async memory read strobe
    awe_l            : IN STD_LOGIC ;        -- EMIF async memory write strobe
    ea_hi            : IN STD_LOGIC_VECTOR(5 DOWNTO 0) ;
                                             -- EMIF Address bits 21:16
    ao_bsy           : IN STD_LOGIC ;        -- Add-On S.M. busy
    emif_ack         : IN STD_LOGIC ;        -- Add-On S.M. EMIF access ack.
    pb_ack           : IN STD_LOGIC ;        -- Peripherial bus access ack.
    xrdy             : IN STD_LOGIC ;        -- EMIF async acc. ready from DB
    ce2sden          : IN STD_LOGIC ;        -- CE2 SDRAM enable from registers
    ce3sden          : IN STD_LOGIC ;        -- CE3 SDRAM enable from registers

    -----------------------------------------------------------------------
    -- OUTPUTS
    -----------------------------------------------------------------------

    dsp2aod_l        : OUT STD_LOGIC ;       -- Enables ED(31:0) to AOD(31:0)
    dsp2gd_l         : OUT STD_LOGIC ;       -- Enables ED(7:0) to GD(31:0)
    dsp2xd_l         : OUT STD_LOGIC ;       -- Enables ED(31:0) to XD(31:0)
awe_pci_l : OUT STD_LOGIC ;   -- awe_l synced to pci clock
are_pci_l : OUT STD_LOGIC ;   -- are_l synced to pci clock
awe_osca_l   : OUT STD_LOGIC ;   -- awe_l synced to clk_a clock
are_osca_l   : OUT STD_LOGIC ;   -- are_l synced to clk_a clock
    emif_req         : OUT STD_LOGIC ;       -- EMIF access request
    falc_req         : OUT STD_LOGIC ;       -- FALC (T1/E1) access request
    fmic_req         : OUT STD_LOGIC ;       -- FMIC access request
    cpld_cs          : OUT STD_LOGIC ;       -- CPLD DSP register chip select
```

```
        ardy            : OUT STD_LOGIC         -- EMIF async access ready

        );
END COMPONENT ;

COMPONENT pb_ctrl
   PORT(
-- active low signals are indicated with ’_l’ appended to signal name.
   -----------------------------------------------------------------------
   -- INPUTS
   -----------------------------------------------------------------------
   brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
   clk_a     : IN STD_LOGIC ; -- Osc A (33.25MHz)
   fmic_req  : IN STD_LOGIC ; -- FMIC decode
   fmic_rdy_in  : IN STD_LOGIC ; -- FMIC ready signal
   falc_req  : IN STD_LOGIC ; -- FALC decode
   lendian      : IN STD_LOGIC ; -- Little endian mode
   be_l      : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ; -- EMIF byte enables
   are_osca_l   : IN STD_LOGIC ; -- are_l synced to clk_a
   awe_osca_l   : IN STD_LOGIC ; -- awe_l synced to clk_a
   -----------------------------------------------------------------------
   -- OUTPUTS
   -----------------------------------------------------------------------
   fmic_cs_l : OUT STD_LOGIC ;   -- FMIC chip select output
   falc_cs_l : OUT STD_LOGIC ;   -- FALC chip select output
   falc_bxe_l   : OUT STD_LOGIC ;   -- FALC BHE/BLE
   falc_a0      : OUT STD_LOGIC ;   -- FALC address 0
   pb_wr_l      : OUT STD_LOGIC ;   -- Peripherial bus write strobe
   pb_rd_l      : OUT STD_LOGIC ;   -- Peripherial bus read strobe
```

```
    pb_ack    : OUT STD_LOGIC  -- state machine done
    );
END COMPONENT ;
COMPONENT irq_ctrl
   PORT(

-- active low signals are indicated with '_l' appended to signal name.

        ------------------------------------------------------------------------
        -- INPUTS
        ------------------------------------------------------------------------

        brd_rst_l       : IN STD_LOGIC ;        -- Reset from volt. super.
        dsp_hint_l      : IN STD_LOGIC ;        -- DSP HPI interrupt
        tbc_int_l       : IN STD_LOGIC ;        -- TBC host interrupt
        pciclk          : IN STD_LOGIC ;        -- Buffered PCI clk (33MHz max)
        pci_det_l       : IN STD_LOGIC ;        -- PCI detection indicator
        pci_irq_l       : IN STD_LOGIC ;        -- Interrupt from add-on device
        rdempty         : IN STD_LOGIC ;        -- PCI read FIFO empty flag
        wrfull          : IN STD_LOGIC ;        -- PCI write FIFO full flag
        hinten          : IN STD_LOGIC ;        -- HPI host interrupt enable
        tbcinten        : IN STD_LOGIC ;        -- TBC host interrupt enable
        dspnmi          : IN STD_LOGIC ;        -- DSP NMI interrupt from host
        nmisel          : IN STD_LOGIC ;        -- NMI source (host/xcvr)
        nmien           : IN STD_LOGIC ;        -- NMI enable
        pcimren         : IN STD_LOGIC ;        -- PCI master read IRQ enable
        pcimwen         : IN STD_LOGIC ;        -- PCI master write IRQ enable
        rdfifo_l        : IN STD_LOGIC ;        -- PCI FIFO read strobe
        wrfifo_l        : IN STD_LOGIC ;        -- PCI FIFO write strobe
        clk_a           : IN STD_LOGIC ;       -- Free-run 33.25 MHZ
    clr_falcint  : IN STD_LOGIC ; -- clear T1/E1 xcvr interrupt
    clr_dbint : IN STD_LOGIC ; -- clear Daughterboard interrupt
    falc_int  : IN STD_LOGIC ; -- T1/E1 xcvr interrupt
    db_int    : IN STD_LOGIC ; -- Daughterboard interrupt

        ----------------------------------------------------------------------
        -- OUTPUTS
        ----------------------------------------------------------------------

    db_falc_int  : OUT STD_LOGIC ;   -- DB/FALC interrupt (EXT_INT7)
    ltchd_falc_int   : OUT STD_LOGIC ;   -- Latched FALC interrupt
    ltchd_db_int  : OUT STD_LOGIC ;   -- Latched DB interrupt
        pc_int          : OUT STD_LOGIC ;       -- Add-on interrupt to PCI cntlr
        pci_int         : OUT STD_LOGIC ;       -- PCI to DSP interrupt (EXT_INT4)
        nmi             : OUT STD_LOGIC ;       -- NMI to DSP
        pcimrd_int      : OUT STD_LOGIC ;       -- PCI mstr read IRQ to DSP (EXT_INT5)
        pcimwr_int      : OUT STD_LOGIC         -- PCI mstr write IRQ to DSP (EXT_INT6)

        );
END COMPONENT ;
COMPONENT misc_glue
   PORT(

-- active low signals are indicated with ’_l’ appended to signal name.

        ----------------------------------------------------------------------
        -- INPUTS
        ----------------------------------------------------------------------
```

```
     -- EVMCKSEL
     clk_a          : IN STD_LOGIC ;       -- Free-run 33.25 MHZ

     -- EVMRESET
     brd_rst_l       : IN STD_LOGIC ;       -- Reset from volt. super.
     sw_rst_l        : IN STD_LOGIC ;       -- Reset from pushbutton
     vcc2bad_l       : IN STD_LOGIC ;       -- VCC low
     pci_rst_l       : IN STD_LOGIC ;       -- Reset from PCI controller
     pci_det_l       : IN STD_LOGIC ;       -- PCI detection (active low)
     xreset          : IN STD_LOGIC ;       -- Ext. reset from DSP register
falcrst      : IN STD_LOGIC ; -- FALC. reset from DSP register
fmicrst      : IN STD_LOGIC ; -- FMIC. reset from DSP register
     dsprst          : IN STD_LOGIC ;       -- DSP reset from PCI register
     tbcrst          : IN STD_LOGIC ;       -- TBC reset from PCI register

     -- EVMSWMUX
     swsel_dip_l     : IN STD_LOGIC ;       -- Switch select
     dip_switch      : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
     soft_switch     : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;

mstr_sel  : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                -- timing mode selection
     fmic_2m_clk     : IN STD_LOGIC ;       -- FMIC 2 MHz clock
     ------------------------------------------------------------------------
     -- OUTPUTS
     ------------------------------------------------------------------------

     -- EVMCKSEL
```

```
        osc_a_en_l      : OUT STD_LOGIC ;       -- Osc. A enable
        osc_b_en_l      : OUT STD_LOGIC ;       -- Osc. B enable

        -- EVMRESET
        man_rst_l       : OUT STD_LOGIC ;       -- Manual reset to volt. super.
        dsp_rst_l       : OUT STD_LOGIC ;       -- DSP reset
        tbc_rst_l       : OUT STD_LOGIC ;       -- TBC reset
        ext_rst_l       : OUT STD_LOGIC ;       -- Daughterboard reset
   falc_rst  : OUT STD_LOGIC ;   -- FALC reset
   fmic_rst_l   : OUT STD_LOGIC ;   -- FMIC reset

   -- To FMIC/FALC timing
   fmic_frm_en_l: OUT STD_LOGIC ;   -- Enable FMIC frame to D.B.
   falc_sync : OUT STD_LOGIC ;   -- FALC sync mux select
        -- EVMSWMUX
        switch          : OUT STD_LOGIC_VECTOR(12 DOWNTO 1)
        );
END COMPONENT ;
--COMPONENT pci_ctrl
COMPONENT pci_control
   PORT(

-- active low signals are indicated with '_l' appended to signal name.

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------

        brd_rst_l       : IN STD_LOGIC ;        -- Reset from volt. super.
```

```
pciclk          : IN STD_LOGIC ;        -- Buffered PCI clk (33MHz max)
clk_mode        : IN STD_LOGIC ;        -- DSP clock mode
pci_det_l       : IN STD_LOGIC ;        -- PCI detection indicator
ptatn_l         : IN STD_LOGIC ;        -- Pass–thru attention signal
ptburst_l       : IN STD_LOGIC ;        -- Pass–thru burst signal
ptnum           : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                                        -- Pass–thru region number
ptwr            : IN STD_LOGIC ;        -- Pass–thru access type (R=0/W=1)
pt_be_l         : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                                        -- Latched pass–thru byte enables
pt_addr         : IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                                        -- Latched pass–thru address
are_pci_l       : IN STD_LOGIC ;        -- sync’d DSP read strobe
awe_pci_l       : IN STD_LOGIC ;        -- sync’d DSP write strobe
be_l            : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                                        -- EMIF byte enables
ea16            : IN STD_LOGIC ;        -- EMIF address bit 16
ea_lo           : IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                                        -- EMIF address bits 6:2
tbc_rdy_l       : IN STD_LOGIC ;        -- JTAG TBC Ready
dsp_hrdy_l      : IN STD_LOGIC ;        -- DSP HPI Ready
emif_req        : IN STD_LOGIC ;        -- Synchronized EMIF access request

------------------------------------------------------------------------
-- OUTPUTS
------------------------------------------------------------------------

pci_flt_l       : OUT STD_LOGIC ;       -- Tri-state PCI controller outputs
pt_adr_l        : OUT STD_LOGIC ;       -- Pass–thru address request
```

```
        pt_rdy_l        : OUT STD_LOGIC ;       -- Pass-thru ready signal
        ao_sel_l        : OUT STD_LOGIC ;       -- Add-on register access
        ao_wr_l         : OUT STD_LOGIC ;       -- Add-on write strobe
        ao_rd_l         : OUT STD_LOGIC ;       -- Add-on read strobe
        ao_adr          : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                                                -- Add-on address bits 6:2
        ao_be_l         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                                                -- Add-on byte enables
        rdfifo_l        : OUT STD_LOGIC ;       -- Read FIFO strobe
        wrfifo_l        : OUT STD_LOGIC ;       -- Write FIFO strobe
        pcireg_oe       : OUT STD_LOGIC ;       -- PCI register output enable
        pcireg_ce       : OUT STD_LOGIC ;       -- PCI register clock enable
        tbc_wr_l        : OUT STD_LOGIC ;       -- TBC write strobe
        tbc_rd_l        : OUT STD_LOGIC ;       -- TBC read strobe
        hcs_l           : OUT STD_LOGIC ;       -- DSP HPI chip select
        hds1_l          : OUT STD_LOGIC ;       -- DSP HPI data strobe
        hrw             : OUT STD_LOGIC ;       -- DSP HPI read/write control
        tbca_hpic       : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                                                -- TBC address bits 4:0/HPI control
        ao_bsy          : OUT STD_LOGIC ;       -- Add-on S.M. busy signal
        emif_ack        : OUT STD_LOGIC         -- EMIF access acknowledge

        );
END COMPONENT ;
COMPONENT registers
   PORT(

-- active low signals are indicated with ’_l’ appended to signal name.
```

```
    -----------------------------------------------------------------------
    -- INPUTS
    -----------------------------------------------------------------------

    dsp_pd          : IN STD_LOGIC ;        -- DSP power down
    vcc2bad_l       : IN STD_LOGIC ;        -- Volt. super. status
    wrfull          : IN STD_LOGIC ;        -- PCI FIFO flag
    rdempty         : IN STD_LOGIC ;        -- PCI FIFO flag
    pcimrd_int      : IN STD_LOGIC ;        -- PCI master read interrupt
    pcimwr_int      : IN STD_LOGIC ;        -- PCI master write interrupt
    tbc_rdy_l       : IN STD_LOGIC ;        -- TBC ready indicator
    tbc_int_l       : IN STD_LOGIC ;        -- TBC interrupt
    dsp_hint_l      : IN STD_LOGIC ;        -- DSP host interrupt
    db_int          : IN STD_LOGIC ;        -- Daughterboard interrupt
    falc_int        : IN STD_LOGIC ;        -- FALC interrupt
ltchd_falc_int   : IN STD_LOGIC ; -- Latched FALC interrupt
ltchd_db_int : IN STD_LOGIC ; -- Latched DB interrupt
    pci_irq_l       : IN STD_LOGIC ;        -- PCI interrupt
    pci_det_l       : IN STD_LOGIC ;        -- PCI detection flag
    cpld_cs         : IN STD_LOGIC ;        -- EMIF acc. 0x0138/0x0178xxxx
    xstat           : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                                            -- Daughterboard status signals
    ea_lo           : IN STD_LOGIC_VECTOR(6 DOWNTO 2) ;
                                            -- lower EMIF adr
    ed              : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                                            -- EMIF data
    pcireg_ce       : IN STD_LOGIC ;        -- PCI register clk enable
    brd_rst_l       : IN STD_LOGIC ;        -- reset from volt. super.
    pciclk          : IN STD_LOGIC ;        -- buffered PCI clockl
```

```
     dq              : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                                             -- PCI cntlr data
     pt_adr_l        : IN STD_LOGIC ;        -- pass thru adr clk enable
     ptbe_l          : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                                             -- latched byte enables
     are_l           : IN STD_LOGIC ;        -- async read enable
     awe_l           : IN STD_LOGIC ;        -- async write enable
     dip_switch      : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                                             -- DIP switch settings
     switch          : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                                             -- selected switch settings
fmic_err  : IN STD_LOGIC ; -- FMIC error signal

     -----------------------------------------------------------------------
     -- OUTPUTS
     -----------------------------------------------------------------------

mstr_sel  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- timing mode selection
ld31_swap : OUT STD_LOGIC ;   -- FMIC stream 3/1 mux ctrl
ld20_swap : OUT STD_LOGIC ;   -- FMIC stream 2/0 mux ctrl
fmic_rst  : OUT STD_LOGIC ;   -- FMIC reset register bit
falc_rst  : OUT STD_LOGIC ;   -- FALC reset register bit
clr_dbint : OUT STD_LOGIC ;   -- clear daughterboard irq
clr_falcint   : OUT STD_LOGIC ;   -- clear FALC irq
loop_led_l    : OUT STD_LOGIC ;   -- drive loop led
sync_led_l    : OUT STD_LOGIC ;   -- drive sync led
ralm_led_l    : OUT STD_LOGIC ;   -- drive red alarm led
yalm_led_l    : OUT STD_LOGIC ;   -- drive yellow alarm led
```

```
a_law_en  : OUT STD_LOGIC ;   -- A–law select
u_law_en  : OUT STD_LOGIC ;   -- u–law select

     ed_oe           : OUT STD_LOGIC ;
     ed_out          : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                                             -- EMIF data out
     dq_out          : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                                             -- PCI data out
     -- DSP mapped register bits
     xreset          : OUT STD_LOGIC ;    -- Daughterboard reset reg. bit
     nmisel          : OUT STD_LOGIC ;    -- NMI selection (xcvr/host)
     nmien           : OUT STD_LOGIC ;    -- NMI enable (maskable NMI?)
     pcimwen         : OUT STD_LOGIC ;    -- PCI master write enable
     pcimren         : OUT STD_LOGIC ;    -- PCI master read enable
     ce2_sden        : OUT STD_LOGIC ;       -- CE2 SDRAM enable
     ce3_sden        : OUT STD_LOGIC ;       -- CE3 SDRAM enable

     -- DSP mapped register bits
     dspnmi          : OUT STD_LOGIC ;    -- Host NMI to DSP
     tbcinten        : OUT STD_LOGIC ;    -- Host TBC interrupt enable
     hinten          : OUT STD_LOGIC ;    -- Host DP host interrupt en.
     tbcrst          : OUT STD_LOGIC ;    -- Host TBC reset reg. bit
     dsprst          : OUT STD_LOGIC ;    -- Host DSP reset reg. bit
     swsel_dip_l     : OUT STD_LOGIC ;    -- DIP or S/W switch selection
     soft_switch     : OUT STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                                             -- Software switches
     pt_addr         : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                                             -- registered pass–thru address
     pt_be_l         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
```

```
                                                -- registered pass-thru byte en.
        xcntl           : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                                                -- Daughterboard control signals
        led             : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                                                -- LED control signals
        sp0sel          : OUT STD_LOGIC      -- DSP serial port select
        );
END COMPONENT ;
BEGIN
-------------------------------------------------------------------------
-- Component instantiations
-------------------------------------------------------------------------
dec_if: decode
   PORT MAP (

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------

        brd_rst_l       => brd_rst_l ,
        pci_det_l       => pci_det_l ,
        pciclk          => bpclk ,
        clk_a           => clk_a ,
        ce1_l           => ce1_l ,
        ce2_l           => ce2_l ,
        ce3_l           => ce3_l ,
        are_l           => are_l ,
        awe_l           => awe_l ,
        ea_hi           => ea_hi ,
        ao_bsy          => ao_bsy ,
        emif_ack        => emif_ack ,
        pb_ack          => pb_ack ,
        xrdy            => xrdy ,
        ce2sden         => ce2sden ,
        ce3sden         => ce3sden ,

        -----------------------------------------------------------------------
        -- OUTPUTS
        -----------------------------------------------------------------------

        dsp2aod_l       => aod_l ,
        dsp2gd_l        => gd_l ,
        dsp2xd_l        => xbd_l ,
        are_pci_l       => are_pci_l ,
        awe_pci_l       => awe_pci_l ,
        are_osca_l      => are_osca_l ,
        awe_osca_l      => awe_osca_l ,
        emif_req        => emif_req ,
        falc_req        => falc_req ,
        fmic_req        => fmic_req ,
        cpld_cs         => cpld_cs ,
        ardy            => async_rdy

        );
pb_if:pb_ctrl
   PORT MAP(
-- active low signals are indicated with '_l' appended to signal name.
    -----------------------------------------------------------------------
    -- INPUTS
    -----------------------------------------------------------------------
    brd_rst_l => brd_rst_l ,
    clk_a     => clk_a ,
    fmic_req  => fmic_req ,
    fmic_rdy_in  => fmic_rdy ,
    falc_req  => falc_req ,
    lendian      => switch(8) ,
    be_l      => be_l ,
    are_osca_l   => are_osca_l ,
    awe_osca_l   => awe_osca_l ,
    -----------------------------------------------------------------------
    -- OUTPUTS
    -----------------------------------------------------------------------
    fmic_cs_l => fmiccs_l ,
    falc_cs_l => falccs_l ,
    falc_bxe_l   => falcbxe_l ,
    falc_a0      => falca0 ,
    pb_wr_l      => pbwr_l ,
    pb_rd_l      => pbrd_l ,
    pb_ack    => pb_ack
    );
irq_if: irq_ctrl
   PORT MAP (

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------
```

```
     brd_rst_l       => brd_rst_l ,
     dsp_hint_l      => dsp_hint_l ,
     tbc_int_l       => tbc_int_l ,
     pciclk          => bpclk ,
     pci_det_l       => pci_det_l ,
     pci_irq_l       => pci_irq_l ,
     rdempty         => rdempty ,
     wrfull          => wrfull ,
     hinten          => hinten ,
     tbcinten        => tbcinten ,
     dspnmi          => dspnmi ,
     nmisel          => nmisel ,
     nmien           => nmien ,
     pcimren         => pcimren ,
     pcimwen         => pcimwen ,
     rdfifo_l        => read_fifo_l ,
     wrfifo_l        => write_fifo_l ,
clk_a     => clk_a ,
clr_falcint  => clr_falcint ,
clr_dbint => clr_dbint ,
     falc_int        => falc_int ,
     db_int          => db_int ,

     -----------------------------------------------------------------------
     -- OUTPUTS
     -----------------------------------------------------------------------

     db_falc_int     => dbfalc_int ,
     ltchd_falc_int  => ltchd_falc_int ,
        ltchd_db_int    => ltchd_db_int ,
        pc_int          => pcint ,
        pci_int         => pciint ,
        nmi             => nmint ,
        pcimrd_int      => pci_mrd_int ,
        pcimwr_int      => pci_mwr_int

        );
misc_if: misc_glue
   PORT MAP (

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------

        clk_a        => clk_a ,

        brd_rst_l       => brd_rst_l ,
        sw_rst_l        => sw_rst_l ,
        vcc2bad_l       => vcc2bad_l ,
        pci_rst_l       => pci_rst_l ,
        pci_det_l       => pci_det_l ,
        xreset          => xreset ,
   falcrst       => falcrst_b ,
   fmicrst       => fmicrst_b ,
        dsprst          => dsprst ,
        tbcrst          => tbcrst ,

        swsel_dip_l     => swsel_dip_l ,
        dip_switch      => dip_switch ,
        soft_switch     => soft_switch ,

   mstr_sel  => mstr_sel ,
   fmic_2m_clk  => fmic_2m_clk ,
        -----------------------------------------------------------------------
        -- OUTPUTS
        -----------------------------------------------------------------------

        osc_a_en_l      => osca_en_l ,
        osc_b_en_l      => oscb_en_l ,

        man_rst_l       => manrst_l ,
        dsp_rst_l       => dsprst_l ,
        tbc_rst_l       => tbcrst_l ,
        ext_rst_l       => xrst_l ,
   falc_rst  => falcrst ,
   fmic_rst_l   => fmicrst_l ,

   -- To FMIC/FALC timing
   fmic_frm_en_l=> fmic_frmen_l ,
   falc_sync => falcsync ,
        switch          => switch
        );
--pci_if: pci_ctrl
pci_if: pci_control
   PORT MAP (

        -----------------------------------------------------------------------
```

```
-- INPUTS
-----------------------------------------------------------------------


brd_rst_l       => brd_rst_l ,
pciclk          => bpclk ,
clk_mode        => switch(6) ,
pci_det_l       => pci_det_l ,
ptatn_l         => ptatn_l ,
ptburst_l       => ptburst_l ,
ptnum           => ptnum ,
ptwr            => ptwr ,
pt_be_l         => pt_be_l ,
pt_addr         => pt_addr ,
are_pci_l       => are_pci_l ,
awe_pci_l       => awe_pci_l ,
be_l            => be_l ,
ea16            => ea16 ,
ea_lo           => ea_lo ,
tbc_rdy_l       => tbc_rdy_l ,
dsp_hrdy_l      => dsp_hrdy_l ,
emif_req        => emif_req ,


-----------------------------------------------------------------------
-- OUTPUTS
-----------------------------------------------------------------------


pci_flt_l       => pciflt_l ,
pt_adr_l        => ptadr_l ,
pt_rdy_l        => ptrdy_l ,
        ao_sel_l        => aosel_l ,
        ao_wr_l         => aowr_l ,
        ao_rd_l         => aord_l ,
        ao_adr          => ao_adr ,
        ao_be_l         => ao_be_l ,
        rdfifo_l        => read_fifo_l ,
        wrfifo_l        => write_fifo_l ,
        pcireg_oe       => pcireg_oe ,
        pcireg_ce       => pcireg_ce ,
        tbc_wr_l        => tbcwr_l ,
        tbc_rd_l        => tbcrd_l ,
        hcs_l           => hpics_l ,
        hds1_l          => hpids_l ,
        hrw             => hpirw ,
        tbca_hpic       => tbca_hpic ,
        ao_bsy          => ao_bsy ,
        emif_ack        => emif_ack

        );
reg_if: registers
   PORT MAP (

        -----------------------------------------------------------------------
        -- INPUTS
        -----------------------------------------------------------------------

        dsp_pd          => dsp_pd ,
        vcc2bad_l       => vcc2bad_l ,
        wrfull          => wrfull ,
     rdempty         => rdempty ,
     pcimrd_int      => pci_mrd_int ,
     pcimwr_int      => pci_mwr_int ,
     tbc_rdy_l       => tbc_rdy_l ,
     tbc_int_l       => tbc_int_l ,
     dsp_hint_l      => dsp_hint_l ,
     db_int          => db_int ,
     falc_int        => falc_int ,
     ltchd_falc_int  => ltchd_falc_int ,
     ltchd_db_int    => ltchd_db_int ,
     pci_irq_l       => pci_irq_l ,
     pci_det_l       => pci_det_l ,
     cpld_cs         => cpld_cs ,
     xstat           => xstat ,
     ea_lo           => ea_lo ,
     ed              => ed ,
     pcireg_ce       => pcireg_ce ,
     brd_rst_l       => brd_rst_l ,
     pciclk          => bpclk ,
     dq              => dq ,
     pt_adr_l        => ptadr_l ,
     ptbe_l          => ptbe_l ,
     are_l           => are_l ,
     awe_l           => awe_l ,
     dip_switch      => dip_switch ,
     switch          => switch ,
fmic_err  => fmic_err ,


     -----------------------------------------------------------------------
     -- OUTPUTS
     ------------------------------------------------------------------------

mstr_sel  => mstr_sel ,
ld31_swap => ld31swap ,
ld20_swap => ld20swap ,
fmic_rst  => fmicrst_b ,
falc_rst  => falcrst_b ,
clr_dbint => clr_dbint ,
clr_falcint   => clr_falcint ,
loop_led_l    => loopled_l ,
sync_led_l    => syncled_l ,
ralm_led_l    => ralmled_l ,
yalm_led_l    => yalmled_l ,
a_law_en  => a_law_en ,
u_law_en  => u_law_en ,
     ed_oe           => ed_oe ,
     ed_out          => ed_out ,
     dq_out          => dq_out ,
     xreset          => xreset ,
     nmisel          => nmisel ,
     nmien           => nmien ,
     pcimwen         => pcimwen ,
     pcimren         => pcimren ,
     ce2_sden        => ce2sden ,
     ce3_sden        => ce3sden ,
     dspnmi          => dspnmi ,
     tbcinten        => tbcinten ,
     hinten          => hinten ,
        tbcrst          => tbcrst ,
        dsprst          => dsprst ,
        swsel_dip_l     => swsel_dip_l ,
        soft_switch     => soft_switch ,
        pt_addr         => pt_addr ,
        pt_be_l         => pt_be_l ,
        xcntl           => xcntl ,
        led             => led ,
        sp0sel          => sp0_sel
        );
-- Concurrent statements
  -- combine individual signals into vectors
  ea_hi(5 DOWNTO 0) <= ea21 & ea20 & ea19 & ea18 & ea17 & ea16 ;
  ea_lo(4 DOWNTO 0) <= ea6 & ea5 & ea4 & ea3 & ea2 ;
  dip_switch(12) <= s_user0 ;
  dip_switch(11) <= s_user1 ;
  dip_switch(10) <= s_user2 ;
  dip_switch(9) <= s_jtagsel ;
  dip_switch(8) <= s_endian ;
  dip_switch(7) <= s_clksel ;
  dip_switch(6) <= s_clkmode ;
  dip_switch(5) <= s_bootmode0 ;
  dip_switch(4) <= s_bootmode1 ;
  dip_switch(3) <= s_bootmode2 ;
  dip_switch(2) <= s_bootmode3 ;
  dip_switch(1) <= s_bootmode4 ;
  ptnum(1 DOWNTO 0) <= ptnum1 & ptnum0 ;
  ptbe_l(3 DOWNTO 0) <= ptbe3_l & ptbe2_l & ptbe1_l & ptbe0_l ;
  be_l(3 DOWNTO 0) <= be3_l & be2_l & be1_l & be0_l ;
```

```
xstat(1 DOWNTO 0) <= xstat1 & xstat0 ;
ed(7 DOWNTO 0) <= ed7 & ed6 & ed5 & ed4 & ed3 & ed2 & ed1 & ed0 ;
dq(7 DOWNTO 0) <= dq7 & dq6 & dq5 & dq4 & dq3 & dq2 & dq1 & dq0 ;
-- tri-state outputs when float_l is low
-- invert led signals to drive proper levels
led1_l  <= ’Z’ WHEN (float_l = true_l) ELSE NOT led(1) ;
led0_l  <= ’Z’ WHEN (float_l = true_l) ELSE NOT led(0) ;
-- control bi-directional signals
ed7     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(7) ;
ed6     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(6) ;
ed5     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(5) ;
ed4     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(4) ;
ed3     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(3) ;
ed2     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(2) ;
ed1     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(1) ;
ed0     <= ’Z’ WHEN ((ed_oe = false_h) OR (float_l = true_l)) ELSE ed_out(0) ;
dq7     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(7) ;
dq6     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(6) ;
dq5     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(5) ;
dq4     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(4) ;
dq3     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(3) ;
dq2     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(2) ;
dq1     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(1) ;
dq0     <= ’Z’ WHEN ((pcireg_oe = false_h) OR (float_l = true_l)) ELSE dq_out(0) ;
pt_adr_l   <= ’Z’ WHEN (float_l = true_l) ELSE ptadr_l ;
ce2_sden   <= ’Z’ WHEN (float_l = true_l) ELSE ce2sden ;
ce3_sden   <= ’Z’ WHEN (float_l = true_l) ELSE ce3sden ;
rdfifo_l   <= ’Z’ WHEN (float_l = true_l) ELSE read_fifo_l ;
wrfifo_l   <= ’Z’ WHEN (float_l = true_l) ELSE write_fifo_l ;
```

```
pcimrd_int <= ’Z’ WHEN (float_l = true_l) ELSE pci_mrd_int ;
pcimwr_int <= ’Z’ WHEN (float_l = true_l) ELSE pci_mwr_int ;
jtagsel    <= ’Z’ WHEN (float_l = true_l) ELSE switch(9) ;
lendian    <= ’Z’ WHEN (float_l = true_l) ELSE switch(8) ;
clkmode    <= ’Z’ WHEN (float_l = true_l) ELSE switch(6) ;
bootmode0  <= ’Z’ WHEN (float_l = true_l) ELSE switch(5) ;
bootmode1  <= ’Z’ WHEN (float_l = true_l) ELSE switch(4) ;
bootmode2  <= ’Z’ WHEN (float_l = true_l) ELSE switch(3) ;
bootmode3  <= ’Z’ WHEN (float_l = true_l) ELSE switch(2) ;
bootmode4  <= ’Z’ WHEN (float_l = true_l) ELSE switch(1) ;
ao_be3_l   <= ’Z’ WHEN (float_l = true_l) ELSE ao_be_l(3) ;
ao_be2_l   <= ’Z’ WHEN (float_l = true_l) ELSE ao_be_l(2) ;
ao_be1_l   <= ’Z’ WHEN (float_l = true_l) ELSE ao_be_l(1) ;
ao_be0_l   <= ’Z’ WHEN (float_l = true_l) ELSE ao_be_l(0) ;
ao_adr6    <= ’Z’ WHEN (float_l = true_l) ELSE ao_adr(4) ;
ao_adr5    <= ’Z’ WHEN (float_l = true_l) ELSE ao_adr(3) ;
ao_adr4    <= ’Z’ WHEN (float_l = true_l) ELSE ao_adr(2) ;
ao_adr3    <= ’Z’ WHEN (float_l = true_l) ELSE ao_adr(1) ;
ao_adr2    <= ’Z’ WHEN (float_l = true_l) ELSE ao_adr(0) ;
tbca4_hcntl1   <= ’Z’ WHEN (float_l = true_l) ELSE tbca_hpic(4) ;
tbca3_hcntl0   <= ’Z’ WHEN (float_l = true_l) ELSE tbca_hpic(3) ;
tbca2_hhwil    <= ’Z’ WHEN (float_l = true_l) ELSE tbca_hpic(2) ;
tbca1_hbe1_l   <= ’Z’ WHEN (float_l = true_l) ELSE tbca_hpic(1) ;
tbca0_hbe0_l   <= ’Z’ WHEN (float_l = true_l) ELSE tbca_hpic(0) ;
xcntl1  <= ’Z’ WHEN (float_l = true_l) ELSE xcntl(1) ;
xcntl0  <= ’Z’ WHEN (float_l = true_l) ELSE xcntl(0) ;
dsp2aod_l  <= ’Z’ WHEN (float_l = true_l) ELSE aod_l ;
dsp2gd_l   <= ’Z’ WHEN (float_l = true_l) ELSE gd_l ;
dsp2xd_l   <= ’Z’ WHEN (float_l = true_l) ELSE xbd_l;
```

```
tbc_wr_l   <= ’Z’ WHEN (float_l = true_l) ELSE tbcwr_l ;
tbc_rd_l   <= ’Z’ WHEN (float_l = true_l) ELSE tbcrd_l ;
tbc_rst_l  <= ’Z’ WHEN (float_l = true_l) ELSE tbcrst_l ;
hcs_l   <= ’Z’ WHEN (float_l = true_l) ELSE hpics_l ;
hds1_l  <= ’Z’ WHEN (float_l = true_l) ELSE hpids_l ;
hrw     <= ’Z’ WHEN (float_l = true_l) ELSE hpirw ;
pci_flt_l  <= ’Z’ WHEN (float_l = true_l) ELSE pciflt_l ;
pt_rdy_l   <= ’Z’ WHEN (float_l = true_l) ELSE ptrdy_l ;
ao_sel_l   <= ’Z’ WHEN (float_l = true_l) ELSE aosel_l ;
ao_wr_l    <= ’Z’ WHEN (float_l = true_l) ELSE aowr_l ;
ao_rd_l    <= ’Z’ WHEN (float_l = true_l) ELSE aord_l;
pc_int  <= ’Z’ WHEN (float_l = true_l) ELSE pcint ;
pci_int    <= ’Z’ WHEN (float_l = true_l) ELSE pciint ;
nmi     <= ’Z’ WHEN (float_l = true_l) ELSE nmint ;
dsp_rst_l  <= ’Z’ WHEN (float_l = true_l) ELSE dsprst_l ;
ardy       <= ’Z’ WHEN (float_l = true_l) ELSE async_rdy ;
osc_a_en_l <= ’Z’ WHEN (float_l = true_l) ELSE osca_en_l ;
osc_b_en_l <= ’Z’ WHEN (float_l = true_l) ELSE oscb_en_l ;
sp0sel  <= ’Z’ WHEN (float_l = true_l) ELSE sp0_sel ;
ext_rst_l  <= ’Z’ WHEN (float_l = true_l) ELSE xrst_l ;
man_rst_l  <= ’Z’ WHEN (float_l = true_l) ELSE manrst_l ;
falc_cs_l  <= ’Z’ WHEN (float_l = true_l) ELSE falccs_l ;
fmic_cs_l  <= ’Z’ WHEN (float_l = true_l) ELSE fmiccs_l ;
falc_moto  <= ’Z’ WHEN (float_l = true_l) ELSE NOT switch(8) ;
falc_bxe_l <= ’Z’ WHEN (float_l = true_l) ELSE falcbxe_l ;
falc_a0    <= ’Z’ WHEN (float_l = true_l) ELSE falca0 ;
pb_wr_l    <= ’Z’ WHEN (float_l = true_l) ELSE pbwr_l ;
pb_rd_l    <= ’Z’ WHEN (float_l = true_l) ELSE pbrd_l ;
ld31_swap  <= ’Z’ WHEN (float_l = true_l) ELSE ld31swap ;
```

```
    ld20_swap  <= ’Z’ WHEN (float_l = true_l) ELSE ld20swap ;
    fmic_rst_l <= ’Z’ WHEN (float_l = true_l) ELSE fmicrst_l ;
    falc_rst   <= ’Z’ WHEN (float_l = true_l) ELSE falcrst ;
    loop_led_l <= ’Z’ WHEN (float_l = true_l) ELSE loopled_l ;
    sync_led_l <= ’Z’ WHEN (float_l = true_l) ELSE syncled_l ;
    ralm_led_l <= ’Z’ WHEN (float_l = true_l) ELSE ralmled_l ;
    yalm_led_l <= ’Z’ WHEN (float_l = true_l) ELSE yalmled_l ;
    alaw_en    <= ’Z’ WHEN (float_l = true_l) ELSE a_law_en ;
    ulaw_en    <= ’Z’ WHEN (float_l = true_l) ELSE u_law_en ;
    fmic_frm_en_l <= ’Z’ WHEN (float_l = true_l) ELSE fmic_frmen_l ;
    falc_sync  <= ’Z’ WHEN (float_l = true_l) ELSE falcsync ;
    db_falc_int    <= ’Z’ WHEN (float_l = true_l) ELSE dbfalc_int ;
  END struct ;
```

```
-------------------------------------------------------------------------
--                                     Design For:
--                Texas Instruments Incorporated
--
--                                     Design By:
--                DNA Enterprises Inc
--                269 West Renner Parkway
--                Richardson, TX 75080
--
--   File name     :  registers.vhd
--   Title         :  Registers
--   Module        :  Top-Level McEVM Control
--   Description   :
--  This module contains all CPLD registers that are written from the
--  DSP or the PCI controller.  These registers were located at the
--  top level of the EVM CPLD Abel source code.
--
--  DSP mapped registers are at base address CE1:0x0138xxxx/0x0178xxxx
--  (MAP0/MAP1).
--
--  DSP Control Register
--  (Address Offset = 0x0, DSP Offset = 0x00)
--    bit 7:  RW:xcntl1
--    bit 6:  RW:xcntl0
--    bit 5:  RW:xreset
--    bit 4:  RW:nmien
--    bit 3:  R :0
--    bit 2:  RW:sp0sel
--    bit 1:  RW:led1
--   bit 0:  RW: led0
-- DSP Status Register
-- (Address Offset = 0x1, DSP Offset = 0x04)
--   bit 7:  R : xstat1
--   bit 6:  R : xstat0
--   bit 5:  R : db_int
--   bit 4:  R : dspnmi
--   bit 3:  R : 0
--   bit 2:  R : 0
--   bit 1:  R : pci_irq_l
--   bit 0:  R : pci_det
-- DSP DIP Switch Options Register
-- (Address Offset = 0x2, DSP Offset = 0x08)
--   bit 7:  R : 0
--   bit 6:  R : s_clkmode
--   bit 5:  R : s_clksel
--   bit 4:  R : s_endian
--   bit 3:  R : s_jtagsel
--   bit 2:  R : s_user2
--   bit 1:  R : s_user1
--   bit 0:  R : s_user0
-- DSP DIP Switch Boot Option Register
-- (Address Offset = 0x3, DSP Offset = 0x0C)
--   bit 7:  R : 0
--   bit 6:  R : 0
--   bit 5:  R : 0
--   bit 4:  R : s_bootmode4
--   bit 3:  R : s_bootmode3
--   bit 2:  R : s_bootmode2
--   bit 1:  R :s_bootmode1
--   bit 0:  R :s_bootmode0
-- DSP Options Register
-- (Address Offset = 0x4, DSP Offset = 0x10)
--   bit 7:  R :0
--   bit 6:  R :clkmode
--   bit 5:  R :clksel
--   bit 4:  R :endian
--   bit 3:  R :jtagsel
--   bit 2:  R :user2
--   bit 1:  R :user1
--   bit 0:  R :user0
-- DSP Boot Option Register
-- (Address Offset = 0x5, DSP Offset = 0x14)
--   bit 7:  R :swsel_dip_l
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :bootmode4
--   bit 3:  R :bootmode3
--   bit 2:  R :bootmode2
--   bit 1:  R :bootmode1
--   bit 0:  R :bootmode0
-- DSP FIFO Status Register
-- (Address Offset = 0x6, DSP Offset = 0x18)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :pcimrd_int
--   bit 4:  R :pcimwr_int
--   bit 3:  R :rdempty
--   bit 2:  R :wrfull
--   bit 1:  RW:pcimren
--   bit 0:  RW:pcimwen
-- DSP SDRAM Control Register
-- (Address Offset = 0x7, DSP Offset = 0x1C)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :0
--   bit 3:  R :0
--   bit 2:  R :0
--   bit 1:  RW:ce3sden
--   bit 0:  RW:ce2sden
-- DSP Oscillator B Frequency
-- (Address Offset = 0x8, DSP Offset = 0x20)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :0
--   bit 3:  R :0
--   bit 2:  R :0
--   bit 1:  R :0
--   bit 0:  R :0  (all zeroes indicates Osc B = 50 MHz)
-- DSP Semaphore 0
-- (Address Offset = 0x9, DSP Offset = 0x24)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :0
--    bit 3:  R :0
--    bit 2:  R :0
--    bit 1:  R :0
--    bit 0:  RW:dsp_sem0
--  DSP Semaphore 1
--  (Address Offset = 0xA, DSP Offset = 0x28)
--    bit 7:  R :0
--    bit 6:  R :0
--    bit 5:  R :0
--    bit 4:  R :0
--    bit 3:  R :0
--    bit 2:  R :0
--    bit 1:  R :0
--    bit 0:  RW:dsp_sem1
--
--  DSP falc Control
--  (Address Offset = 0xB, DSP Offset = 0x2C)
--    bit 7:  RW:mstr_sel3
--    bit 6:  RW:mstr_sel2
--    bit 5:  RW:mstr_sel1
--    bit 4:  RW:mstr_sel0
--    bit 3:  RW:ld31_swap
--    bit 2:  RW:ld20_swap
--    bit 1:  RW:fmic_rst
--    bit 0:  RW:falc_rst
--
--  DSP Interrupt Control
--  (Address Offset = 0xC, DSP Offset = 0x30)
--    bit 7:  R :ltchd_db_int
--   bit 6:  R :ltchd_falc_int
--   bit 5:  R :db_int
--   bit 4:  R :falc_int
--   bit 3:  R :0
--   bit 2:  R :0
--   bit 1:  RW:clr_dbint
--   bit 0:  RW:clr_falcint
--
-- DSP Miscellaneous Status
-- (Address Offset = 0xD, DSP Offset = 0x34)
--   bit 7:  RW:loop_led
--   bit 6:  RW:sync_led
--   bit 5:  RW:red_alm_led
--   bit 4:  RW:yel_alm_led
--   bit 3:  RW:u_a_law
--   bit 2:  RW:vbap_en
--   bit 1:  R :fmic_err
--   bit 0:  R :0
--
-- PCI mapped registers are at base address BAR2.  Actual address is
-- base + (offset * 4)
--
-- PCI Control Register
-- (Offset = 0x0, PCI Offset = 0x00)
--   bit 7:  RW:dspnmi
--   bit 6:  RW:tbcinten
--   bit 5:  RW:hinten
--   bit 4:  R :tbc_rdy
--   bit 3:  R :tbc_int
--   bit 2:  R :dsp_hint
--   bit 1:  RW:tbcrst
--   bit 0:  RW:dsprst
-- PCI Status Register
-- (Offset = 0x1, PCI Offset = 0x04)
--   bit 7:  R :0
--   bit 6:  R :vcc2bad
--   bit 5:  R :0
--   bit 4:  R :dsp_pd
--   bit 3:  R :xcntl1
--   bit 2:  R :xcntl0
--   bit 1:  R :led1
--   bit 0:  R :led0
-- PCI Software Switch Option Register
-- (Offset = 0x2, PCI Offset = 0x08)
--   bit 7:  R :0
--   bit 6:  RW:h_clkmode
--   bit 5:  RW:h_clksel
--   bit 4:  RW:h_endian
--   bit 3:  RW:h_jtagsel
--   bit 2:  RW:h_user2
--   bit 1:  RW:h_user1
--   bit 0:  RW:h_user0
-- PCI Software Switch Boot Option Register
-- (Offset = 0x3, PCI Offset = 0x0C)
--   bit 7:  RW:swsel_dip_l
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  RW:h_bootmode4
--   bit 3:  RW:h_bootmode3
--   bit 2:  RW:h_bootmode2
--   bit 1:  RW:h_bootmode1
--   bit 0:  RW:h_bootmode0
-- PCI DIP Switch Option Register
-- (Offset = 0x4, PCI Offset = 0x10)
--   bit 7:  R :0
--   bit 6:  R :s_clkmode
--   bit 5:  R :s_clksel
--   bit 4:  R :s_endian
--   bit 3:  R :s_jtagsel
--   bit 2:  R :s_user2
--   bit 1:  R :s_user1
--   bit 0:  R :s_user0
-- PCI DIP Switch Boot Option Register
-- (Offset = 0x5, PCI Offset = 0x14)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :s_bootmode4
--   bit 3:  R :s_bootmode3
--   bit 2:  R :s_bootmode2
--   bit 1:  R :s_bootmode1
--   bit 0:  R :s_bootmode0
-- PCI Option Register
-- (Offset = 0x6, PCI Offset = 0x18)
--   bit 7:  R :0
--   bit 6:  R :clkmode
--   bit 5:  R :clksel
--   bit 4:  R :endian
--   bit 3:  R :jtagsel
--   bit 2:  R :user2
--   bit 1:  R :user1
--   bit 0:  R :user0
-- PCI Boot Option Register
-- (Offset = 0x7, PCI Offset = 0x1C)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :bootmode4
--   bit 3:  R :bootmode3
--   bit 2:  R :bootmode2
--   bit 1:  R :bootmode1
--   bit 0:  R :bootmode0
-- PCI CPLD Revision Register
-- (Offset = 0x8, PCI Offset = 0x20)
--   bit 7:  R :CPLD revision bit 7
--   bit 6:  R :CPLD revision bit 6
--   bit 5:  R :CPLD revision bit 5
--   bit 4:  R :CPLD revision bit 4
--   bit 3:  R :CPLD revision bit 3
--   bit 2:  R :CPLD revision bit 2
--   bit 1:  R :CPLD revision bit 1
--   bit 0:  R :CPLD revision bit 0
-- PCI Semaphore 0
-- (Address Offset = 0x9, PCI Offset = 0x24)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :0
--   bit 3:  R :0
--   bit 2:  R :0
--   bit 1:  R :0
--   bit 0:  RW:pci_sem0
-- PCI Semaphore 1
-- (Address Offset = 0xA, PCI Offset = 0x25)
--   bit 7:  R :0
--   bit 6:  R :0
--   bit 5:  R :0
--   bit 4:  R :0
--   bit 3:  R :0
--   bit 2:  R :0
--   bit 1:  R :0
--   bit 0:  RW:pci_sem1
-------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     03/31/98
-- Author:      Don Curry (DNA)
-- Description: Initial conversion from ABEL version of EVM CPLD.
--
-- Revision: 1
-- Date:     05/01/98
-- Author:      Don Curry (DNA)
-- Description: Added PCI and DSP semaphore register bits as well
--        as the OSC_B_FREQ register.
```

```
--
--  Revision: 2
--  Date:     05/03/98
--  Author:      Don Curry (DNA)
--  Description:  Changed NMISEL to always select host (0) since
--          codec is no longer valid. Added McEVM specific
--          registers.
--
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY registers IS
   PORT(
-- active low signals are indicated with '_l' appended to signal name.
   -----------------------------------------------------------------------
   -- INPUTS
   -----------------------------------------------------------------------
   dsp_pd    : IN STD_LOGIC ; -- DSP power down
   vcc2bad_l : IN STD_LOGIC ; -- Volt. super. status
   wrfull    : IN STD_LOGIC ; -- PCI FIFO flag
   rdempty      : IN STD_LOGIC ; -- PCI FIFO flag
   pcimrd_int   : IN STD_LOGIC ; -- PCI master read interrupt
   pcimwr_int   : IN STD_LOGIC ; -- PCI master write interrupt
   tbc_rdy_l : IN STD_LOGIC ; -- TBC ready indicator
   tbc_int_l : IN STD_LOGIC ; -- TBC interrupt
   dsp_hint_l   : IN STD_LOGIC ; -- DSP host interrupt
   db_int    : IN STD_LOGIC ; -- Daughterboard interrupt
```

```
falc_int  : IN STD_LOGIC ; -- FALC interrupt
ltchd_falc_int   : IN STD_LOGIC ; -- Latched FALC interrupt
ltchd_db_int  : IN STD_LOGIC ; -- Latched DB interrupt
pci_irq_l : IN STD_LOGIC ; -- PCI interrupt
pci_det_l : IN STD_LOGIC ; -- PCI detection flag
cpld_cs       : IN STD_LOGIC ; -- EMIF acc. 0x0138/0x0178xxxx
xstat     : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                 -- Daughterboard status signals
ea_lo     : IN STD_LOGIC_VECTOR(6 DOWNTO 2) ;
                 -- lower EMIF adr
ed     : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                 -- EMIF data
pcireg_ce : IN STD_LOGIC ; -- PCI register clk enable
brd_rst_l : IN STD_LOGIC ; -- reset from volt. super.
pciclk    : IN STD_LOGIC ; -- buffered PCI clockl
dq     : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                 -- PCI cntlr data
pt_adr_l  : IN STD_LOGIC ; -- pass thru adr clk enable
ptbe_l    : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- latched byte enables
are_l     : IN STD_LOGIC ; -- async read enable
awe_l     : IN STD_LOGIC ; -- async write enable
dip_switch    : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                 -- DIP switch settings
switch    : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                 -- selected switch settings
fmic_err  : IN STD_LOGIC ; -- FMIC error signal
------------------------------------------------------------------------
-- OUTPUTS
```

```
-----------------------------------------------------------------------
ed_oe     : OUT STD_LOGIC ;
ed_out    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                 -- EMIF data out
dq_out    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                 -- PCI data out
-- DSP mapped register bits
xreset    : OUT STD_LOGIC ;   -- Daughterboard reset reg. bit
nmisel    : OUT STD_LOGIC ;   -- NMI selection (host)
nmien     : OUT STD_LOGIC ;   -- NMI enable (maskable NMI?)
pcimwen       : OUT STD_LOGIC ;   -- PCI master write enable
pcimren       : OUT STD_LOGIC ;   -- PCI master read enable
ce2_sden  : OUT STD_LOGIC ;   -- CE2 SDRAM enable
ce3_sden  : OUT STD_LOGIC ;   -- CE3 SDRAM enable
-- McEVM specific control bits
mstr_sel  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- timing mode selection
ld31_swap : OUT STD_LOGIC ;   -- FMIC stream 3/1 mux ctrl
ld20_swap : OUT STD_LOGIC ;   -- FMIC stream 2/0 mux ctrl
fmic_rst  : OUT STD_LOGIC ;   -- FMIC reset register bit
falc_rst  : OUT STD_LOGIC ;   -- FALC reset register bit
clr_dbint : OUT STD_LOGIC ;   -- clear daughterboard irq
clr_falcint   : OUT STD_LOGIC ;   -- clear FALC irq
loop_led_l    : OUT STD_LOGIC ;   -- drive loop led
sync_led_l    : OUT STD_LOGIC ;   -- drive sync led
ralm_led_l    : OUT STD_LOGIC ;   -- drive red alarm led
yalm_led_l    : OUT STD_LOGIC ;   -- drive yellow alarm led
a_law_en  : OUT STD_LOGIC ;   -- A-law select
u_law_en  : OUT STD_LOGIC ;   -- u-law select
```

```
   -- PCI mapped register bits
   dspnmi    : OUT STD_LOGIC ;   -- Host NMI to DSP
   tbcinten  : OUT STD_LOGIC ;   -- Host TBC interrupt enable
   hinten    : OUT STD_LOGIC ;   -- Host DP host interrupt en.
   tbcrst    : OUT STD_LOGIC ;   -- Host TBC reset reg. bit
   dsprst    : OUT STD_LOGIC ;   -- Host DSP reset reg. bit
   swsel_dip_l  : OUT STD_LOGIC ;   -- DIP or S/W switch selection
   soft_switch  : OUT STD_LOGIC_VECTOR(12 DOWNTO 1) ;
                    -- Software switches
   pt_addr      : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                    -- registered pass-thru address
   pt_be_l      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                    -- registered pass-thru byte en.
   xcntl     : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    -- Daughterboard control signals
   led    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    -- LED control signals
   sp0sel    : OUT STD_LOGIC  -- DSP serial port select
   );
END registers ;
ARCHITECTURE rtl OF registers IS
-- Define types used
-- Define constants for readability
  CONSTANT true_h    : STD_LOGIC := '1' ;
  CONSTANT false_h   : STD_LOGIC := '0' ;
  CONSTANT true_l    : STD_LOGIC := '0' ;
  CONSTANT false_l   : STD_LOGIC := '1' ;

  -- Revision control
```

```
CONSTANT cpld_rev : STD_LOGIC_VECTOR(7 DOWNTO 0) := ″00000000″ ;
CONSTANT osc_b_freq  : STD_LOGIC_VECTOR(7 DOWNTO 0) := ″00101000″ ;
–– reserved register values
CONSTANT pci_rsvd : STD_LOGIC_VECTOR(7 DOWNTO 0) := ″10111100″ ;
CONSTANT dsp_rsvd : STD_LOGIC_VECTOR(7 DOWNTO 0) := ″00000000″ ;
–– DSP Address definitions
CONSTANT dsp_cntl_adr   : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00000″ ; –– 0x0
CONSTANT dsp_stat_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00001″ ; –– 0x1
CONSTANT dsp_dipopt_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00010″ ; –– 0x2
CONSTANT dsp_dipboot_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00011″ ; –– 0x3
CONSTANT dsp_dspopt_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00100″ ; –– 0x4
CONSTANT dsp_dspboot_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00101″ ; –– 0x5
CONSTANT dsp_fifostat_adr: STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00110″ ; –– 0x6
CONSTANT dsp_sdcntl_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00111″ ; –– 0x7
CONSTANT dsp_oscb_fq_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01000″ ; –– 0x8
CONSTANT dsp_sem0_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01001″ ; –– 0x9
CONSTANT dsp_sem1_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01010″ ; –– 0xA
CONSTANT dsp_falcctrl_adr: STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01011″ ; –– 0xB
CONSTANT dsp_intctrl_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01100″ ; –– 0xC
CONSTANT dsp_miscctrl_adr: STD_LOGIC_VECTOR(4 DOWNTO 0) := ″01101″ ; –– 0xD
–– PCI Address definitions
CONSTANT pci_cntl_adr   : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00000″ ; –– 0x0
CONSTANT pci_stat_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00001″ ; –– 0x1
CONSTANT pci_swopt_adr   : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00010″ ; –– 0x2
CONSTANT pci_swboot_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00011″ ; –– 0x3
CONSTANT pci_dipopt_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00100″ ; –– 0x4
CONSTANT pci_dipboot_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00101″ ; –– 0x5
CONSTANT pci_dspopt_adr  : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00110″ ; –– 0x6
CONSTANT pci_dspboot_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ″00111″ ; –– 0x7
```

```
  CONSTANT pci_cpldrev_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ”01000” ; -- 0x8
  CONSTANT pci_sem0_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ”01001” ; -- 0x9
  CONSTANT pci_sem1_adr    : STD_LOGIC_VECTOR(4 DOWNTO 0) := ”01010” ; -- 0xA
-- Internal signal declarations
  -- software switches
  SIGNAL h_clkmode  : STD_LOGIC ;-- Clock mode S/W switch
  SIGNAL h_clksel   : STD_LOGIC ;-- Clock selection S/W switch
  SIGNAL h_endian   : STD_LOGIC ;-- Endiam mode S/W switch
  SIGNAL h_jtagsel  : STD_LOGIC ;-- JTAG selection S/W switch
  SIGNAL h_user2 : STD_LOGIC ;-- User defined S/W switch
  SIGNAL h_user1 : STD_LOGIC ;-- User defined S/W switch
  SIGNAL h_user0 : STD_LOGIC ;-- User defined S/W switch
  SIGNAL h_bootmode4   : STD_LOGIC ;-- Bootmode bit S/W switch
  SIGNAL h_bootmode3   : STD_LOGIC ;-- Bootmode bit S/W switch
  SIGNAL h_bootmode2   : STD_LOGIC ;-- Bootmode bit S/W switch
  SIGNAL h_bootmode1   : STD_LOGIC ;-- Bootmode bit S/W switch
  SIGNAL h_bootmode0   : STD_LOGIC ;-- Bootmode bit S/W switch
  -- DSP register vectors
  SIGNAL dsp_cntl   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_stat   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_dipopt    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_dipboot   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_dspopt    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_dspboot   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_fifostat  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_sdcntl    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_oscb_fq   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_sem_0  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL dsp_sem_1  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL dsp_falcctrl  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL dsp_intctrl   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL dsp_miscctrl  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL mstr_sel3  : STD_LOGIC ;
SIGNAL mstr_sel2  : STD_LOGIC ;
SIGNAL mstr_sel1  : STD_LOGIC ;
SIGNAL mstr_sel0  : STD_LOGIC ;
SIGNAL st31_swap  : STD_LOGIC ;
SIGNAL st20_swap  : STD_LOGIC ;
SIGNAL fmicrst : STD_LOGIC ;
SIGNAL falcrst : STD_LOGIC ;
SIGNAL clrdbint   : STD_LOGIC ;
SIGNAL clrfalcint : STD_LOGIC ;
SIGNAL loop_led   : STD_LOGIC ;
SIGNAL sync_led   : STD_LOGIC ;
SIGNAL ralm_led   : STD_LOGIC ;
SIGNAL yalm_led   : STD_LOGIC ;
SIGNAL alaw_sel   : STD_LOGIC ;
SIGNAL vbap_en : STD_LOGIC ;
-- PCI register vectors
SIGNAL pci_cntl   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_stat   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_swopt  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_swboot    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_dipopt    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_dipboot   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_dspopt    : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_dspboot   : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL pci_sem_0  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
```

```
  SIGNAL pci_sem_1  : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  -- SDRAM enables
  SIGNAL ce2sden : STD_LOGIC ;
  SIGNAL ce3sden : STD_LOGIC ;
  -- internal signals for passing to ports
  SIGNAL xrst   : STD_LOGIC ;
--  SIGNAL nmi_sel  : STD_LOGIC ;    -- R2
  SIGNAL nmi_en     : STD_LOGIC ;
  SIGNAL pci_mwen   : STD_LOGIC ;
  SIGNAL pci_mren   : STD_LOGIC ;
  SIGNAL dsp_nmi : STD_LOGIC ;
  SIGNAL tbc_inten  : STD_LOGIC ;
  SIGNAL hpi_inten  : STD_LOGIC ;
  SIGNAL tbc_rst : STD_LOGIC ;
  SIGNAL dsp_rst : STD_LOGIC ;
  SIGNAL sw_sel_dip_l  : STD_LOGIC ;
  SIGNAL sw_switch  : STD_LOGIC_VECTOR(12 DOWNTO 1) ;
  SIGNAL pt_adr     : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
  SIGNAL xctrl      : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL led_ctrl   : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL sp0_sel : STD_LOGIC ;
  SIGNAL dsp_sem0_ar_l : STD_LOGIC ;
  SIGNAL pci_sem0_ar_l : STD_LOGIC ;
  SIGNAL dsp_semaphore0: STD_LOGIC ;
  SIGNAL pci_semaphore0: STD_LOGIC ;
  SIGNAL dsp_sem1_ar_l : STD_LOGIC ;
  SIGNAL pci_sem1_ar_l : STD_LOGIC ;
  SIGNAL dsp_semaphore1: STD_LOGIC ;
  SIGNAL pci_semaphore1: STD_LOGIC ;
```

```
BEGIN
-- Pass internal siganls to port outputs
  xreset <= xrst ;
  nmisel <= false_h ;      -- R2
  nmien <= nmi_en ;
  pcimwen <= pci_mwen ;
  pcimren <= pci_mren ;
  dspnmi <= dsp_nmi ;
  tbcinten <= tbc_inten ;
  hinten <= hpi_inten ;
  tbcrst <= tbc_rst ;
  dsprst <= dsp_rst ;
  swsel_dip_l <= sw_sel_dip_l ;
  soft_switch <= sw_switch ;
  pt_addr <= pt_adr ;
  xcntl <= xctrl ;
  led <= led_ctrl ;
  sp0sel <= sp0_sel ;
  mstr_sel(3) <= mstr_sel3 ;
  mstr_sel(2) <= mstr_sel2 ;
  mstr_sel(1) <= mstr_sel1 ;
  mstr_sel(0) <= mstr_sel0 ;
  ld31_swap <= st31_swap ;
  ld20_swap <= st20_swap ;
  fmic_rst <= fmicrst ;
  falc_rst <= falcrst ;
  clr_dbint <= clrdbint ;
  clr_falcint <= clrfalcint ;
  loop_led_l <= NOT loop_led ;
  sync_led_l <= NOT sync_led ;
  ralm_led_l <= NOT ralm_led ;
  yalm_led_l <= NOT yalm_led ;
  -- map S/W switches to vector in same order as DIP switch bits
  sw_switch(12) <= h_user0 ;
  sw_switch(11) <= h_user1 ;
  sw_switch(10) <= h_user2 ;
  sw_switch(9) <= h_jtagsel ;
  sw_switch(8) <= h_endian ;
  sw_switch(7) <= h_clksel ;
  sw_switch(6) <= h_clkmode ;
  sw_switch(5) <= h_bootmode0 ;
  sw_switch(4) <= h_bootmode1 ;
  sw_switch(3) <= h_bootmode2 ;
  sw_switch(2) <= h_bootmode3 ;
  sw_switch(1) <= h_bootmode4 ;
-- Concurrent statements
  -- map VBAP en/a-law to external signals
  a_law_en <= true_h WHEN ((vbap_en = true_h) AND
                           (alaw_sel = true_h)) ELSE
              false_h ;

  u_law_en <= true_h WHEN ((vbap_en = true_h) AND
                           (alaw_sel = false_h)) ELSE
              false_h ;

  -- qualify ce2_sden with reset
  ce2_sden <= true_h WHEN ((ce2sden = true_h) AND
                           (brd_rst_l = false_l)) ELSE
```

```
         false_h ;
  -- qualify ce3_sden with reset
  ce3_sden <= true_h WHEN ((ce3sden = true_h) AND
                           (brd_rst_l = false_l)) ELSE
         false_h ;
  -- DSP reads
  ed_oe <= true_h WHEN ((cpld_cs = true_h) AND
                        (are_l = true_l)) ELSE
      false_h ;
--ed_out <= dsp_cntl AND (ea_lo = ”00001”)
--     OR dsp_stat AND (ea_lo = ”00001”)
--     OR dsp_dipopt AND (ea_lo = ”00010”)
--     OR dsp_dipboot AND (ea_lo = ”00011”)
--     OR dsp_dspopt AND (ea_lo = ”00100”)
--     OR dsp_dspboot AND (ea_lo = ”00101”)
--     OR dsp_fifostat AND (ea_lo = ”00110”)
--     OR dsp_sdcntl AND (ea_lo = ”00111”)
--     OR osc_b_freq AND (ea_lo = ”01000”)
--     OR dsp_sem_0 AND (ea_lo = ”01001”)
--     OR dsp_sem_1 AND (ea_lo = ”01010”)
--     OR dsp_falcctrl AND (ea_lo = ”01011”)
--     OR dsp_intctrl AND (ea_lo = ”01100”)
--     OR dsp_miscctrl AND (ea_lo = ”01101”)
--     OR dsp_rsvd;
  WITH ea_lo SELECT -- ea_lo is 5-bit input
    ed_out <= dsp_cntl WHEN dsp_cntl_adr, -- dsp_cntl_addr = ”00000”
         dsp_stat      WHEN dsp_stat_adr, -- dsp_stat_adr = ”00001”
         dsp_dipopt    WHEN dsp_dipopt_adr, -- dsp_dipopt_adr = ”00010”
         dsp_dipboot   WHEN dsp_dipboot_adr, -- dsp_dipboot_adr = ”00011”
```

```
       dsp_dspopt    WHEN dsp_dspopt_adr, -- dsp_dspopt_adr = ”00100”
       dsp_dspboot   WHEN dsp_dspboot_adr, -- dsp_dspboot_adr = ”00101”
       dsp_fifostat  WHEN dsp_fifostat_adr, -- dsp_fifostat_adr = ”00110”
       dsp_sdcntl    WHEN dsp_sdcntl_adr, -- dsp_sdcntl_adr = ”00111”
       osc_b_freq    WHEN dsp_oscb_fq_adr, -- dsp_oscb_fq_adr = ”01000”
       dsp_sem_0  WHEN dsp_sem0_adr, -- dsp_sem0_adr = ”01001”
       dsp_sem_1  WHEN dsp_sem1_adr, -- dsp_sem1_adr = ”01010”
       dsp_falcctrl  WHEN dsp_falcctrl_adr, -- dsp_falcctrl_adr = ”01011”
       dsp_intctrl   WHEN dsp_intctrl_adr, -- dsp_intctrl_adr = ”01100”
       dsp_miscctrl  WHEN dsp_miscctrl_adr, -- dsp_miscctrl_adr = ”01101”
       dsp_rsvd      WHEN OTHERS ; -- dsp_rsvd  = ”00000000”
-- PCI reads
WITH pt_adr SELECT
  dq_out <= pci_cntl     WHEN pci_cntl_adr, -- pci_cntl_adr = ”00000”
       pci_stat      WHEN pci_stat_adr, -- pci_stat_adr = ”00001”
       pci_swopt  WHEN pci_swopt_adr, -- pci_swopt_adr = ”00010”
       pci_swboot    WHEN pci_swboot_adr, -- pci_swboot_adr = ”00011”
       pci_dipopt    WHEN pci_dipopt_adr, -- pci_dipopt_adr = ”00100”
       pci_dipboot   WHEN pci_dipboot_adr, -- pci_dipboot_adr = ”00101”
       pci_dspopt    WHEN pci_dspopt_adr, -- pci_dspopt_adr = ”00110”
       pci_dspboot   WHEN pci_dspboot_adr, -- pci_dspboot_adr = ”00111”
       cpld_rev      WHEN pci_cpldrev_adr, -- pci_cpldrev_adr = ”01000”
       pci_sem_0  WHEN pci_sem0_adr, -- pci_sem0_adr = ”01001”
       pci_sem_1  WHEN pci_sem1_adr, -- pci_sem1_adr = ”01010”
       pci_rsvd      WHEN OTHERS ; -- pci_rsvd = ”10111100”
-- collect DSP read back bits by concatenating signals into vectors
dsp_cntl(7 DOWNTO 6) <= xctrl(1 DOWNTO 0) ;
dsp_cntl(5) <= xrst ;
dsp_cntl(4) <= nmi_en ;
```

```
--  dsp_cntl(3) <= nmi_sel ;     -- R2
  dsp_cntl(3) <= ’0’ ;
  dsp_cntl(2) <= sp0_sel ;
  dsp_cntl(1 DOWNTO 0) <= led_ctrl(1 DOWNTO 0) ;
  dsp_stat(7 DOWNTO 6) <= xstat(1 DOWNTO 0) ;
  dsp_stat(5) <= db_int ;
  dsp_stat(4) <= dsp_nmi ;
  dsp_stat(3 DOWNTO 2) <= ”00” ;
  dsp_stat(1) <= NOT pci_irq_l ;
  dsp_stat(0) <= NOT pci_det_l ;
  dsp_dipopt(7) <= ’0’ ;
  dsp_dipopt(6) <= dip_switch(6) ;
  dsp_dipopt(5) <= dip_switch(7) ;
  dsp_dipopt(4) <= dip_switch(8) ;
  dsp_dipopt(3) <= dip_switch(9) ;
  dsp_dipopt(2) <= dip_switch(10) ;
  dsp_dipopt(1) <= dip_switch(11) ;
  dsp_dipopt(0) <= dip_switch(12) ;
  dsp_dipboot(7 DOWNTO 5) <= ”000” ;
  dsp_dipboot(4) <= dip_switch(1) ;
  dsp_dipboot(3) <= dip_switch(2) ;
 dsp_dipboot(2) <= dip_switch(3) ;
  dsp_dipboot(1) <= dip_switch(4) ;
  dsp_dipboot(0) <= dip_switch(5) ;
  dsp_dspopt(7) <= ’0’ ;
  dsp_dspopt(6) <= switch(6) ;
  dsp_dspopt(5) <= switch(7) ;
  dsp_dspopt(4) <= switch(8) ;
  dsp_dspopt(3) <= switch(9) ;
dsp_dspopt(2) <= switch(10) ;
dsp_dspopt(1) <= switch(11) ;
dsp_dspopt(0) <= switch(12) ;
dsp_dspboot(7) <= sw_sel_dip_l ;
dsp_dspboot(6 DOWNTO 5) <= ”00” ;
dsp_dspboot(4) <= switch(1) ;
dsp_dspboot(3) <= switch(2) ;
dsp_dspboot(2) <= switch(3) ;
dsp_dspboot(1) <= switch(4) ;
dsp_dspboot(0) <= switch(5) ;
dsp_fifostat(7 DOWNTO 6) <= ”00” ;
dsp_fifostat(5) <= pcimrd_int ;
dsp_fifostat(4) <= pcimwr_int ;
dsp_fifostat(3) <= rdempty ;
dsp_fifostat(2) <= wrfull ;
dsp_fifostat(1) <= pci_mren ;
dsp_fifostat(0) <= pci_mwen ;
dsp_sdcntl(7 DOWNTO 2) <= ”000000” ;
dsp_sdcntl(1) <= ce3sden ;
dsp_sdcntl(0) <= ce2sden ;
dsp_sem_0(7 DOWNTO 1) <= ”0000000” ;
dsp_sem_0(0) <= dsp_semaphore0 ;
dsp_sem_1(7 DOWNTO 1) <= ”0000000” ;
dsp_sem_1(0) <= dsp_semaphore1 ;
dsp_falcctrl(7) <= mstr_sel3 ;
dsp_falcctrl(6) <= mstr_sel2 ;
dsp_falcctrl(5) <= mstr_sel1 ;
dsp_falcctrl(4) <= mstr_sel0 ;
dsp_falcctrl(3) <= st31_swap ;
dsp_falcctrl(2) <= st20_swap ;
dsp_falcctrl(1) <= fmicrst ;
dsp_falcctrl(0) <= falcrst ;
dsp_intctrl(7) <= ltchd_db_int ;
dsp_intctrl(6) <= ltchd_falc_int ;
dsp_intctrl(5) <= db_int ;
dsp_intctrl(4) <= falc_int ;
dsp_intctrl(3) <= ’0’ ;
dsp_intctrl(2) <= ’0’ ;
dsp_intctrl(1) <= clrdbint ;
dsp_intctrl(0) <= clrfalcint ;
dsp_miscctrl(7) <= loop_led ;
dsp_miscctrl(6) <= sync_led ;
dsp_miscctrl(5) <= ralm_led ;
dsp_miscctrl(4) <= yalm_led ;
dsp_miscctrl(3) <= alaw_sel ;
dsp_miscctrl(2) <= vbap_en ;
dsp_miscctrl(1) <= fmic_err ;
dsp_miscctrl(0) <= ’0’ ;

-- collect PCI read back bits by concatenating signals into vectors
pci_cntl(7) <= dsp_nmi ;
pci_cntl(6) <= tbc_inten ;
pci_cntl(5) <= hpi_inten ;
pci_cntl(4) <= NOT tbc_rdy_l ;
pci_cntl(3) <= NOT tbc_int_l ;
pci_cntl(2) <= NOT dsp_hint_l ;
pci_cntl(1) <= tbc_rst ;
pci_cntl(0) <= dsp_rst ;
```

```
pci_stat(7) <= ’0’ ;
pci_stat(6) <= NOT vcc2bad_l ;
pci_stat(5) <= ’0’ ;
pci_stat(4) <= dsp_pd ;
pci_stat(3 DOWNTO 2) <= xctrl(1 DOWNTO 0) ;
pci_stat(1 DOWNTO 0) <= led_ctrl(1 DOWNTO 0) ;
pci_swopt(7) <= ’0’ ;
pci_swopt(6) <= sw_switch(6) ;
pci_swopt(5) <= sw_switch(7) ;
pci_swopt(4) <= sw_switch(8) ;
pci_swopt(3) <= sw_switch(9) ;
pci_swopt(2) <= sw_switch(10) ;
pci_swopt(1) <= sw_switch(11) ;
pci_swopt(0) <= sw_switch(12) ;
pci_swboot(7) <= sw_sel_dip_l ;
pci_swboot(6 DOWNTO 5) <= ”00” ;
pci_swboot(4) <= sw_switch(1) ;
pci_swboot(3) <= sw_switch(2) ;
pci_swboot(2) <= sw_switch(3) ;
pci_swboot(1) <= sw_switch(4) ;
pci_swboot(0) <= sw_switch(5) ;
pci_dipopt(7) <= ’0’ ;
pci_dipopt(6) <= dip_switch(6) ;
pci_dipopt(5) <= dip_switch(7) ;
pci_dipopt(4) <= dip_switch(8) ;
pci_dipopt(3) <= dip_switch(9) ;
pci_dipopt(2) <= dip_switch(10) ;
pci_dipopt(1) <= dip_switch(11) ;
pci_dipopt(0) <= dip_switch(12) ;
```

```
pci_dipboot(7 DOWNTO 5) <= ”000” ;
pci_dipboot(4) <= dip_switch(1) ;
pci_dipboot(3) <= dip_switch(2) ;
pci_dipboot(2) <= dip_switch(3) ;
pci_dipboot(1) <= dip_switch(4) ;
pci_dipboot(0) <= dip_switch(5) ;
pci_dspopt(7) <= ’0’ ;
pci_dspopt(6) <= switch(6) ;
pci_dspopt(5) <= switch(7) ;
pci_dspopt(4) <= switch(8) ;
pci_dspopt(3) <= switch(9) ;
pci_dspopt(2) <= switch(10) ;
pci_dspopt(1) <= switch(11) ;
pci_dspopt(0) <= switch(12) ;
pci_dspboot(7 DOWNTO 5) <= ”000” ;
pci_dspboot(4) <= switch(1) ;
pci_dspboot(3) <= switch(2) ;
pci_dspboot(2) <= switch(3) ;
pci_dspboot(1) <= switch(4) ;
pci_dspboot(0) <= switch(5) ;
pci_sem_0(7 DOWNTO 1) <= ”0000000” ;
pci_sem_0(0) <= pci_semaphore0 ;
pci_sem_1(7 DOWNTO 1) <= ”0000000” ;
pci_sem_1(0) <= pci_semaphore1 ;
–– Semaphore async reset signal generation
dsp_sem0_ar_l <= true_l WHEN ((brd_rst_l = true_l) OR
                              (pci_semaphore0 = true_h)) ELSE
                 false_l ;
dsp_sem1_ar_l <= true_l WHEN ((brd_rst_l = true_l) OR
                                (pci_semaphore1 = true_h)) ELSE
                   false_l ;
  pci_sem0_ar_l <= true_l WHEN ((brd_rst_l = true_l) OR
                                (dsp_semaphore0 = true_h)) ELSE
                   false_l ;
  pci_sem1_ar_l <= true_l WHEN ((brd_rst_l = true_l) OR
                                (dsp_semaphore1 = true_h)) ELSE
                   false_l ;
-- Sequential statements
-- Semaphore implementation
-- Note when one side obtains the semphore, the other side is prevented
-- from getting it by holding the semaphore FF in reset.  This forces
-- the losing side to re-write the semaphore to try to get it as opposed
-- to just polling it.  Done this way to maintain S/W compatibility with
-- EVM.
dspsem0:PROCESS(awe_l, dsp_sem0_ar_l)
BEGIN
  IF (dsp_sem0_ar_l = true_l) THEN
    dsp_semaphore0 <= false_h ;
  ELSIF ((awe_l’EVENT) AND (awe_l = false_l)) THEN

    IF (cpld_cs = true_h) THEN
      IF (ea_lo = dsp_sem0_adr) THEN
   dsp_semaphore0 <= ed(0) ;
      END IF ;
    END IF ;
  END IF ;   -- clocked IF
END PROCESS dspsem0 ;
dspsem1:PROCESS(awe_l, dsp_sem1_ar_l)
```

```
BEGIN
  IF (dsp_sem1_ar_l = true_l) THEN
    dsp_semaphore1 <= false_h ;
  ELSIF ((awe_l’EVENT) AND (awe_l = false_l)) THEN

    IF (cpld_cs = true_h) THEN
      IF (ea_lo = dsp_sem1_adr) THEN
   dsp_semaphore1 <= ed(0) ;
      END IF ;
    END IF ;
  END IF ;   -- clocked IF
END PROCESS dspsem1 ;
pcisem0:PROCESS(pciclk, pci_sem0_ar_l)
BEGIN
  IF (pci_sem0_ar_l = true_l) THEN
    pci_semaphore0 <= false_h ;
  ELSIF ((pciclk’EVENT) AND (pciclk = false_l)) THEN

    IF (pcireg_ce = true_h) THEN
      IF (pt_adr = pci_sem0_adr) THEN
   pci_semaphore0 <= dq(0) ;
      END IF ;
    END IF ;
  END IF ;   -- clocked IF
END PROCESS pcisem0 ;
pcisem1:PROCESS(pciclk, pci_sem1_ar_l)
BEGIN
  IF (pci_sem1_ar_l = true_l) THEN
    pci_semaphore1 <= false_h ;
  ELSIF ((pciclk’EVENT) AND (pciclk = false_l)) THEN

    IF (pcireg_ce = true_h) THEN
      IF (pt_adr = pci_sem1_adr) THEN
   pci_semaphore1 <= dq(0) ;
      END IF ;
    END IF ;
  END IF ;   -- clocked IF
END PROCESS pcisem1 ;
-- DSP writes are done by awe_l when cpld_cs is active.
dspwrt:PROCESS(awe_l, brd_rst_l)
BEGIN
  IF (brd_rst_l = true_l) THEN
    xctrl <= ”00” ;    -- External control set to 0’s
    led_ctrl <= ”00” ;     -- led’s off
    xrst <= false_h ;      -- Ext reset de–asserted
    nmi_en <= false_h ;    -- NMI disabled
--    nmi_sel <= false_h ; -- NMI from host    -- R2
    sp0_sel <= false_h ;   -- McBSP0 to daughterboard
    pci_mren <= false_h ;  -- PCI bus master read disabled
    pci_mwen <= false_h ;  -- PCI bus master write disabled
    ce3sden <= true_h ;    -- CE3 SDRAM enabled
    ce2sden <= true_h ;    -- CE2 SDRAM enabled
    mstr_sel3 <= false_h ; -- Master select = 0000
    mstr_sel2 <= false_h ;
    mstr_sel1 <= false_h ;
    mstr_sel0 <= false_h ;
    st31_swap <= false_h ; -- No swap FMIC ST busses 3 and 1
    st20_swap <= false_h ; -- No swap FMIC ST busses 2 and 0
    fmicrst <= false_h ;   -- No FMIC reset
    falcrst <= false_h ;   -- No FALC reset
    clrdbint <= false_h ; -- No clear DB interrupt (int. enabled)
    clrfalcint <= false_h ;   -- No clear FALC interrupt (int. enabled)
    loop_led <= false_h ; -- LED off
    sync_led <= false_h ; -- LED off
    ralm_led <= false_h ; -- LED off
    yalm_led <= false_h ; -- LED off
    alaw_sel <= false_h ; -- u-law default
    vbap_en <= false_h ;   -- VBAP's disabled
  ELSIF ((awe_l'EVENT) AND (awe_l = false_l)) THEN

    IF (cpld_cs = true_h) THEN
      CASE ea_lo IS
   WHEN dsp_cntl_adr =>
     xctrl   <= ed(7 DOWNTO 6) ;
     xrst       <= ed(5) ;
     nmi_en  <= ed(4) ;
--   nmi_sel    <= ed(3) ;       -- R2
     sp0_sel    <= ed(2) ;
     led_ctrl   <= ed(1 DOWNTO 0) ;
   WHEN dsp_fifostat_adr =>
     pci_mren   <= ed(1) ;
     pci_mwen   <= ed(0) ;
   WHEN dsp_sdcntl_adr =>
     ce3sden    <= ed(1) ;
     ce2sden    <= ed(0) ;
   WHEN dsp_falcctrl_adr =>
          mstr_sel3 <= ed(7) ;
          mstr_sel2 <= ed(6) ;
          mstr_sel1 <= ed(5) ;
          mstr_sel0 <= ed(4) ;
          st31_swap <= ed(3) ;
          st20_swap <= ed(2) ;
          fmicrst   <= ed(1) ;
          falcrst     <= ed(0) ;
   WHEN dsp_intctrl_adr =>
          clrdbint <= ed(1) ;
          clrfalcint <= ed(0) ;
   WHEN dsp_miscctrl_adr =>
          loop_led <= ed(7) ;
          sync_led <= ed(6) ;
          ralm_led <= ed(5) ;
          yalm_led <= ed(4) ;
          alaw_sel <= ed(3) ;
          vbap_en  <= ed(2) ;
   WHEN OTHERS => NULL ;
      END CASE ;
    END IF ; -- cpld_cs
  END IF ;   -- clocked IF
END PROCESS dspwrt ;
pci_pt:PROCESS(pciclk, brd_rst_l)
-- register pass through address and byte enables during strobe
BEGIN
  IF (brd_rst_l = true_l) THEN
    pt_adr <= ”11111” ;
    pt_be_l <= ”1111” ;
  ELSIF ((pciclk’EVENT) AND (pciclk = true_h)) THEN
```

```
    IF (pt_adr_l = true_l) THEN
      pt_adr <= dq(6 DOWNTO 2) ;
      pt_be_l <= ptbe_l ;
    END IF ;
  END IF ;   -- clocked IF
END PROCESS pci_pt ;
pciwrt:PROCESS(pciclk, brd_rst_l)
-- PCI writes
BEGIN
  IF (brd_rst_l = true_l) THEN
    dsp_nmi <= false_h ;   -- No NMI
    tbc_inten <= false_h ; -- TBC interrupt disabled
    hpi_inten <= false_h ; -- DSP host interrupt disabled
    tbc_rst <= false_h ;   -- TBC reset de–asserted
    dsp_rst <= false_h ;   -- DSP reset de–asserted
    h_clkmode <= false_h ; -- x4 PLL mode
    h_clksel <= false_h ;  -- Oscillator A (33.25 MHz)
    h_endian <= false_h ;  -- Little endian
    h_jtagsel <= false_h ; -- External JTAG
    h_user2 <= false_h ;   -- User switch2 = 0
    h_user1 <= false_h ;   -- User switch1 = 0
    h_user0 <= false_h ;   -- User switch0 = 0
    sw_sel_dip_l <= true_l ;  -- DIP switch selected
    -- bootmode defaulted to NO–BOOT, Internal, MAP1 (mode 5)
    h_bootmode4 <= false_h ;
    h_bootmode3 <= false_h ;
    h_bootmode2 <= true_h ;
    h_bootmode1 <= false_h ;
    h_bootmode0 <= true_h ;
  ELSIF ((pciclk’EVENT) AND (pciclk = true_h)) THEN
    IF (pcireg_ce = true_h) THEN
      CASE pt_adr IS
   WHEN pci_cntl_adr =>
     dsp_nmi    <= dq(7) ;
     tbc_inten  <= dq(6) ;
     hpi_inten  <= dq(5) ;
     tbc_rst    <= dq(1) ;
     dsp_rst    <= dq(0) ;
   WHEN pci_swopt_adr =>
     h_clkmode  <= dq(6) ;
     h_clksel   <= dq(5) ;
     h_endian   <= dq(4) ;
     h_jtagsel  <= dq(3) ;
     h_user2    <= dq(2) ;
     h_user1    <= dq(1) ;
     h_user0    <= dq(0) ;
   WHEN pci_swboot_adr =>
     sw_sel_dip_l   <= dq(7) ;
     h_bootmode4    <= dq(4) ;
     h_bootmode3    <= dq(3) ;
     h_bootmode2    <= dq(2) ;
     h_bootmode1    <= dq(1) ;
     h_bootmode0    <= dq(0) ;
   WHEN OTHERS => NULL ;
      END CASE ;
    END IF ; –– pcireg_ce IF
  END IF ;   –– clocked IF
END PROCESS pciwrt ;
END rtl ;
```

```
-------------------------------------------------------------------------
--                                    Design For:
--                        Texas Instruments Incorporated
--
--                                    Design By:
--              DNA Enterprises Inc
--              269 West Renner Parkway
--              Richardson, TX 75080
--
--   File name     :  pb_ctrl.vhd
--   Title         :  McEVM Peripherial Controller
--   Module        :  Top–Level McEVM Control
--   Description   :
-- This module provides a state machine to control DSP accesses to
-- the FALC and the FMIC devices.  The state machine assures that
-- proper timing is generated on the control signals to those parts.
--
-------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     05/04/98
-- Author:      Don Curry (DNA)
-- Description: Initial code.
--
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY pb_ctrl IS
   PORT(
-- active low signals are indicated with ’_l’ appended to signal name.
    ------------------------------------------------------------------------
    -- INPUTS
    ------------------------------------------------------------------------
    brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
    clk_a     : IN STD_LOGIC ; -- Osc A (33.25MHz)
    lendian       : IN STD_LOGIC ; -- little endian mode
    be_l      : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;  -- DSP byte en
    are_osca_l    : IN STD_LOGIC ; -- are_l synced to clk_a
    awe_osca_l    : IN STD_LOGIC ; -- awe_l synced to clk_a
    falc_req  : IN STD_LOGIC ; -- FALC decode
    fmic_req  : IN STD_LOGIC ; -- FMIC decode
    fmic_rdy_in   : IN STD_LOGIC ; -- FMIC ready signal
    ------------------------------------------------------------------------
    -- OUTPUTS
    ------------------------------------------------------------------------
    fmic_cs_l : OUT STD_LOGIC ;   -- FMIC chip select output
    falc_cs_l : OUT STD_LOGIC ;   -- FALC chip select output
    falc_bxe_l    : OUT STD_LOGIC ;   -- FALC BHE/BLE
    falc_a0       : OUT STD_LOGIC ;   -- FALC address 0
    pb_wr_l       : OUT STD_LOGIC ;   -- Peripherial bus write strobe
    pb_rd_l       : OUT STD_LOGIC ;   -- Peripherial bus read strobe
    pb_ack    : OUT STD_LOGIC  -- state machine done
    );
END pb_ctrl ;
```

```
ARCHITECTURE rtl OF pb_ctrl IS
-- Define types used
  TYPE pb_state_type IS (idle,
                         fmic_s1, fmic_s2, fmic_s3, fmic_rd,
                         fmic_s1a, fmic_s2a, fmic_s3a, fmic_rda,
                         falc_s1, falc_s1a,
                         falc_le_s2, falc_le_s3,
                         falc_le_s2a, falc_le_s3a,
                         falc_be_s2, falc_be_s3,
                         falc_be_s2a, falc_be_s3a,
                         ack) ;
-- Define constants for readability
  CONSTANT true_h   : STD_LOGIC := ’1’ ;
  CONSTANT false_h  : STD_LOGIC := ’0’ ;
  CONSTANT true_l   : STD_LOGIC := ’0’ ;
  CONSTANT false_l  : STD_LOGIC := ’1’ ;
-- Internal signal declarations
  SIGNAL pb_state   : pb_state_type ;
  SIGNAL pb_next_state : pb_state_type ;
  SIGNAL pbrd_l     : STD_LOGIC ;
  SIGNAL pback      : STD_LOGIC ;
  SIGNAL falccs_l   : STD_LOGIC ;
  SIGNAL fmiccs_l   : STD_LOGIC ;
  SIGNAL fmic_rdy   : STD_LOGIC_VECTOR(1 DOWNTO 0) ;

 BEGIN
-- Pass internal siganls to port outputs
-- Concurrent statements
  falc_bxe_l <= be_l(1) WHEN (lendian = true_h) ELSE
```

```
                be_l(0) ;
  falc_a0 <= be_l(0) WHEN (lendian = true_h) ELSE
             be_l(1) ;
  falc_cs_l <= falccs_l ;
  fmic_cs_l <= fmiccs_l ;

  -- shut down read signal ASAP when sync read goes away to keep from being
  -- on bus when DSP starts next access.
  pb_rd_l <= true_l WHEN ((pbrd_l = true_l) AND
                          ((are_osca_l = true_l) OR  -- im = 0
                           (awe_osca_l = true_l))) ELSE  -- im = 1
             false_l ;

  pb_ack <= pback  ;

-- Sequential statements
pb_sm:PROCESS(pb_state, fmic_req, falc_req, lendian,
              awe_osca_l, are_osca_l, fmic_rdy(1))
BEGIN
  CASE pb_state IS
  ------------------------------------------------------------------------
    WHEN idle => -- wait for sync'd read or write strobe active
  ------------------------------------------------------------------------
      IF ((are_osca_l = true_l) OR
          (awe_osca_l = true_l)) THEN
        IF (fmic_req = true_h) THEN
          pb_next_state <= fmic_s1 ;
        ELSIF (falc_req = true_h) THEN
          pb_next_state <= falc_s1 ;
      ELSE
        pb_next_state <= pb_state ;
      END IF ;
    ELSE
      pb_next_state <= pb_state ;
    END IF ;
------------------------------------------------------------------------
  WHEN fmic_s1 =>
------------------------------------------------------------------------
    pb_next_state <= fmic_s1a ;
------------------------------------------------------------------------
  WHEN fmic_s1a =>
------------------------------------------------------------------------
    pb_next_state <= fmic_s2 ;
------------------------------------------------------------------------
  WHEN fmic_s2 =>
------------------------------------------------------------------------
    pb_next_state <= fmic_s2a ;
------------------------------------------------------------------------
  WHEN fmic_s2a =>
------------------------------------------------------------------------
    pb_next_state <= fmic_s3 ;
------------------------------------------------------------------------
  WHEN fmic_s3 =>
------------------------------------------------------------------------
    pb_next_state <= fmic_s3a ;
------------------------------------------------------------------------
  WHEN fmic_s3a =>
------------------------------------------------------------------------
```

```
    IF (fmic_rdy(1) = true_h) THEN
      IF (awe_osca_l = true_l) THEN
        pb_next_state <= ack ;
      ELSE
        pb_next_state <= fmic_rd ;
      END IF ;
    ELSE
      pb_next_state <= pb_state ;
    END IF ;
------------------------------------------------------------------------
  WHEN fmic_rd =>
------------------------------------------------------------------------
    pb_next_state <= fmic_rda ;
------------------------------------------------------------------------
  WHEN fmic_rda =>
------------------------------------------------------------------------
    pb_next_state <= ack ;
  -- FALC part
------------------------------------------------------------------------
  WHEN falc_s1 =>
------------------------------------------------------------------------
    pb_next_state <= falc_s1a ;
------------------------------------------------------------------------
  WHEN falc_s1a =>
------------------------------------------------------------------------
    IF (lendian = true_h) THEN
      pb_next_state <= falc_le_s2 ;
    ELSE
      pb_next_state <= falc_be_s2 ;
    END IF ;
------------------------------------------------------------------------
  WHEN falc_le_s2 =>
------------------------------------------------------------------------
    pb_next_state <= falc_le_s2a ;
------------------------------------------------------------------------
  WHEN falc_le_s2a =>
------------------------------------------------------------------------
    pb_next_state <= falc_le_s3 ;
------------------------------------------------------------------------
  WHEN falc_le_s3 =>
------------------------------------------------------------------------
    pb_next_state <= falc_le_s3a ;
------------------------------------------------------------------------
  WHEN falc_le_s3a =>
------------------------------------------------------------------------
    pb_next_state <= ack ;
------------------------------------------------------------------------
  WHEN falc_be_s2 =>
------------------------------------------------------------------------
    pb_next_state <= falc_be_s2a ;
------------------------------------------------------------------------
  WHEN falc_be_s2a =>
------------------------------------------------------------------------
    pb_next_state <= falc_be_s3 ;
------------------------------------------------------------------------
  WHEN falc_be_s3 =>
------------------------------------------------------------------------
    pb_next_state <= falc_be_s3a ;
  ------------------------------------------------------------------------
    WHEN falc_be_s3a =>
  ------------------------------------------------------------------------
      pb_next_state <= ack ;
  ------------------------------------------------------------------------
    WHEN ack =>
  ------------------------------------------------------------------------
      IF ((awe_osca_l = false_l) AND
          (are_osca_l = false_l)) THEN
        pb_next_state <= idle ;
      ELSE
        pb_next_state <= pb_state ;
      END IF ;
  ------------------------------------------------------------------------
    WHEN OTHERS =>
  ------------------------------------------------------------------------
      pb_next_state <= idle ;
  END CASE ;
END PROCESS pb_sm ;
seq:PROCESS(clk_a, brd_rst_l)
BEGIN
  IF (brd_rst_l = true_l) THEN
    pb_state <= idle ;
    pbrd_l <= false_l ;
    pb_wr_l <= false_l ;
    fmiccs_l <= false_l ;
    fmic_rdy <= ”00” ;
    falccs_l <= false_l ;
    pback <= false_h ;
```

```
ELSIF ((clk_a’EVENT) AND
       (clk_a = true_h)) THEN

  pb_state <= pb_next_state ;

  -- de-metastablize FMIC ready signal
  fmic_rdy(0) <= fmic_rdy_in ;
  fmic_rdy(1) <= fmic_rdy(0) ;

  CASE pb_state IS
------------------------------------------------------------------------
    WHEN idle =>
------------------------------------------------------------------------
      IF (fmic_req = true_h) THEN
        fmiccs_l <= true_l ;
      ELSIF (falc_req = true_h) THEN
        falccs_l <= true_l ;
      END IF ;
      IF ((falc_req = true_h) AND
          (lendian = false_h) AND
          (awe_osca_l = true_l)) THEN
        pb_wr_l <= true_l ;
      END IF ;
------------------------------------------------------------------------
    WHEN fmic_s1 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN fmic_s1a =>
```

```
------------------------------------------------------------------------
      IF (awe_osca_l = true_l) THEN
        pb_wr_l <= true_l ;
      ELSIF (are_osca_l = true_l) THEN
        pbrd_l <= true_l ;
      END IF ;
------------------------------------------------------------------------
    WHEN fmic_s2 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN fmic_s2a =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN fmic_s3 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN fmic_s3a =>
------------------------------------------------------------------------
      IF ((fmic_rdy(1) = true_h) AND
          (awe_osca_l = true_l)) THEN
            pb_wr_l <= false_l ;
            pback <= true_h ;
      END IF ;
------------------------------------------------------------------------
    WHEN fmic_rd =>
------------------------------------------------------------------------
```

```
      NULL ;
------------------------------------------------------------------------
    WHEN fmic_rda =>
------------------------------------------------------------------------
      pback <= true_h ;
    -- FALC part
------------------------------------------------------------------------
    WHEN falc_s1 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_s1a =>
------------------------------------------------------------------------
      IF (lendian = true_h) THEN
        IF (awe_osca_l = true_l) THEN
          pb_wr_l <= true_l ;
        ELSIF (are_osca_l = true_l) THEN
          pbrd_l <= true_l ;
        END IF ;
      ELSE
        pbrd_l <= true_l ;
      END IF ;
------------------------------------------------------------------------
    WHEN falc_le_s2 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_le_s2a =>
------------------------------------------------------------------------
```

```
      NULL ;
------------------------------------------------------------------------
    WHEN falc_le_s3 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_le_s3a =>
------------------------------------------------------------------------
      IF (awe_osca_l = true_l) THEN
        pb_wr_l <= false_l ;
      END IF ;
      pback <= true_h ;
------------------------------------------------------------------------
    WHEN falc_be_s2 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_be_s2a =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_be_s3 =>
------------------------------------------------------------------------
      NULL ;
------------------------------------------------------------------------
    WHEN falc_be_s3a =>
------------------------------------------------------------------------
      IF (awe_osca_l = true_l) THEN
        pbrd_l <= false_l ;
        END IF ;
        pback <= true_h ;
  ------------------------------------------------------------------------
      WHEN ack =>
  ------------------------------------------------------------------------
        IF ((awe_osca_l = false_l) AND
            (are_osca_l = false_l)) THEN
          pback <= false_h ;
          pb_wr_l <= false_l ;
          pbrd_l <= false_l ;
          fmiccs_l <= false_l ;
          falccs_l <= false_l ;
        END IF ;
  ------------------------------------------------------------------------
      WHEN OTHERS =>
  ------------------------------------------------------------------------
        pback <= false_h ;
        pbrd_l <= false_l ;
        pb_wr_l <= false_l ;
        fmiccs_l <= false_l ;
        falccs_l <= false_l ;
    END CASE ;
  END IF ; -- clocked IF
END PROCESS seq ;

END rtl ;
```

```
-------------------------------------------------------------------------
--                                     Design For:
--                        Texas Instruments Incorporated
--
--                                     Design By:
--               DNA Enterprises Inc
--               269 West Renner Parkway
--               Richardson, TX 75080
--
--   File name     :  misc_glue.vhd
--   Title         :  Miscellaneous Glue
--   Module        :  Top–Level McEVM Control
--   Description   :
-- This module combines the EVMCKSEL, EVMRESET and EVMSWMUX functions
-- from the EVM CPLD Abel files.
--
-- The EVMCKSEL implements DSP source clock selection. The DSP source
-- clock (CLKIN) can either be 33.25MHz or 50MHz. The two oscillator
-- outputs are buffered using buffers with independent output enables.
-- The outputs of these buffers are connected together. This module
-- makes sure that there is never any contention on this signal by
-- turning off both buffers for one clock period prior to turning on
-- one. The clock used for this function is the free–running clock from
-- Osc. A (33.25 MHz).
--
-- The EVMRESET controls the various types of reset signals that are
-- used on the McEVM board.  Signals are generated that reset the board,
-- DSP, JTAG TBC and external daughterboard independently. All resets are
-- asserted on power-up, when the manual reset pushbutton is pressed and
-- when the board is in a software board reset under control of the host.
-- The host can reset the board, DSP and TBC. The DSP can reset the
-- external daughterboard.
--
-- The EVMSWMUX provides a multiplexer to switch between hardware DIP
-- switches and host software controllable registers for various modes.
-- The CPLD defaults to the DIP switches at power-up or board reset. The
-- multiplexer is controlled by the host software using the PCI mapped
-- CPLD SWBOOT registers SWSEL bit. A 0 (default) selects the DIP switch
-- and a 1 selects the software register bits.
--
------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     03/30/98
-- Author:      Don Curry (DNA)
-- Description: Initial conversion from ABEL version of EVM CPLD.
--
------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY misc_glue IS
   PORT(
-- active low signals are indicated with '_l' appended to signal name.
   ----------------------------------------------------------------------
```

```
-- INPUTS
------------------------------------------------------------------------
-- EVMCKSEL
clk_a     : IN STD_LOGIC ; -- Free–run 33.25 MHz
-- EVMRESET
brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
sw_rst_l  : IN STD_LOGIC ; -- Reset from pushbutton
vcc2bad_l : IN STD_LOGIC ; -- VCC low
pci_rst_l : IN STD_LOGIC ; -- Reset from PCI controller
pci_det_l : IN STD_LOGIC ; -- PCI detection (active low)
xreset    : IN STD_LOGIC ; -- Ext. reset from DSP register
falcrst       : IN STD_LOGIC ; -- FALC reset from DSP register
fmicrst       : IN STD_LOGIC ; -- FMIC reset from DSP register
dsprst    : IN STD_LOGIC ; -- DSP reset from PCI register
tbcrst    : IN STD_LOGIC ; -- TBC reset from PCI register
-- EVMSWMUX
swsel_dip_l   : IN STD_LOGIC ; -- Switch select
dip_switch    : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
soft_switch   : IN STD_LOGIC_VECTOR(12 DOWNTO 1) ;
mstr_sel  : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- timing mode selection
     fmic_2m_clk     : IN STD_LOGIC ;        -- FMIC 2 MHz clock
------------------------------------------------------------------------
-- OUTPUTS
------------------------------------------------------------------------
-- EVMCKSEL
osc_a_en_l    : OUT STD_LOGIC ;   -- Osc. A enable
osc_b_en_l    : OUT STD_LOGIC ;   -- Osc. B enable
-- EVMRESET
```

```
    man_rst_l : OUT STD_LOGIC ;   -- Manual reset to volt. super.
    dsp_rst_l : OUT STD_LOGIC ;   -- DSP reset
    tbc_rst_l : OUT STD_LOGIC ;   -- TBC reset
    ext_rst_l : OUT STD_LOGIC ;   -- Daughterboard reset
    falc_rst  : OUT STD_LOGIC ;   -- FALC reset
    fmic_rst_l    : OUT STD_LOGIC ;   -- FMIC reset
    -- To FMIC/FALC timing
    fmic_frm_en_l: OUT STD_LOGIC ;   -- Enable FMIC frame to D.B.
    falc_sync : OUT STD_LOGIC ;   -- FALC sync mux output
    -- EVMSWMUX
    switch    : OUT STD_LOGIC_VECTOR(12 DOWNTO 1)
                    -- Selected switch setting
    );
END misc_glue ;
ARCHITECTURE rtl OF misc_glue IS
-- Define types used
-- Define constants for readability
  CONSTANT true_h   : STD_LOGIC := '1' ;
  CONSTANT false_h  : STD_LOGIC := '0' ;
  CONSTANT true_l   : STD_LOGIC := '0' ;
  CONSTANT false_l  : STD_LOGIC := '1' ;
-- Internal signal declarations
  SIGNAL sel_switch : STD_LOGIC_VECTOR(12 DOWNTO 1) ;
  SIGNAL clksel_db1 : STD_LOGIC ;-- clock select delayed by 1 clock
  SIGNAL clksel_db2 : STD_LOGIC ;-- clock select delayed by 2 clock
BEGIN
-- Pass internal siganls to port outputs
  switch <= sel_switch ;
-- Concurrent statements
```

```
---------------------------------------------------------------------------
-- EVMCKSEL
---------------------------------------------------------------------------
-- Osc A (33.25 MHz) selected when clksel is low for two consecutive clocks
osc_a_en_l <= true_l WHEN ((clksel_db1 = false_h) AND
         (clksel_db2 = false_h)) ELSE
      false_l ;
-- Osc B (50 MHz) selected when clksel is high for two consecutive clocks
osc_b_en_l <= true_l WHEN ((clksel_db1 = true_h) AND
         (clksel_db2 = true_h)) ELSE
      false_l ;
---------------------------------------------------------------------------
-- EVMRESET
---------------------------------------------------------------------------
-- Manual reset to voltage supervisor asserted when PCI reset or pushbutton
-- is activated. Note that pushbutton will bounce, however the voltage
-- supervisor will assert reset on first occurance and hold reset active
-- until approximately 140 ns after last detected low on input thus providing
-- a de-bounce function.
man_rst_l <= true_l WHEN (((pci_rst_l = true_l) AND (pci_det_l = true_l)) OR
           (sw_rst_l = true_l)) ELSE
       false_l ;
-- DSP reset asserted when DSP software reset is asserted, DSP core voltage
-- is bad or a board reset. Allows the host to control DSP reset for HPI
-- booting.
dsp_rst_l <= true_l WHEN ((dsprst = true_h) OR
           (vcc2bad_l = true_l) OR
           (brd_rst_l = true_l)) ELSE
        false_l ;
```

```
-- TBC reset asserted upon host assertion or board reset.
tbc_rst_l <= true_l WHEN ((tbcrst = true_h) OR
           (brd_rst_l = true_l)) ELSE
        false_l ;
-- External reset asserted upon DSP assertion or board reset.
ext_rst_l <= true_l WHEN ((xreset = true_h) OR
           (brd_rst_l = true_l)) ELSE
        false_l ;
-- FALC reset asserted upon DSP assertion or board reset.
falc_rst <= true_h WHEN ((falcrst = true_h) OR
           (brd_rst_l = true_l)) ELSE
       false_h ;
-- FMIC reset asserted upon DSP assertion or board reset.
fmic_rst_l <= true_l WHEN ((fmicrst = true_h) OR
             (brd_rst_l = true_l)) ELSE
         false_l ;
---------------------------------------------------------------------------
-- EVMSWMUX
---------------------------------------------------------------------------
-- switch bit mapping is as follows
--   BIT SIGNAL
--   1:5 bootmode4:bootmode0 DSP bootmode
--   6   clkmode          DSP clock mode x1/x4
--   7   clksel       DSP osc. select 33.25/50 MHz
--   8   endian       Little/Big endian mode
--   9   jtagsel          Internal/External JTAG emulation
--   10:12 user2:user0      User defined switches
sel_switch <= soft_switch(12 DOWNTO 9) &
     -- invert to drive correct polarity to DSP
```

```
                NOT soft_switch(8) &
                soft_switch(7) &
       -- invert to drive correct polarity to DSP
                NOT soft_switch(6) &
                soft_switch(5 DOWNTO 1) WHEN (swsel_dip_l = true_h) ELSE
           dip_switch(12 DOWNTO 9) &
       -- invert to drive correct polarity to DSP
                NOT dip_switch(8) &
                dip_switch(7) &
       -- invert to drive correct polarity to DSP
                NOT dip_switch(6) &
                dip_switch(5 DOWNTO 1) ;
  WITH mstr_sel SELECT
      falc_sync <= fmic_2m_clk WHEN ”0001” | ”0011” ,
                   false_h     WHEN OTHERS ;
  WITH mstr_sel SELECT
      fmic_frm_en_l <= true_l WHEN ”0010” | ”0011” ,
                       false_l WHEN OTHERS ;
-- Sequential statements
evmcksel:PROCESS(clk_a)
-- Note: The selected clock must be enabled during reset so that the DSP
-- PLL locks before reset is released.  This means the FF’s should not
-- be held in reset.
-- Switch mux clksel output is fed into two cascaded FF’s to generate a
-- break–before–make clock switch.
BEGIN
  IF ((clk_a’EVENT) AND
     (clk_a = true_h)) THEN
```

```
    clksel_db1 <= sel_switch(7) ;-- Clock select bit
    clksel_db2 <= clksel_db1 ;
  END IF ; -- clocked IF
END PROCESS evmcksel ;

END rtl ;
```

```
-------------------------------------------------------------------------
--                                     Design For:
--                         Texas Instruments Incorporated
--
--                                     Design By:
--               DNA Enterprises Inc
--               269 West Renner Parkway
--               Richardson, TX 75080
--
--   File name     :  decode.vhd
--   Title         :  McEVM Async Memory Space Decode
--   Module        :  Top–Level McEVM Control
--   Description   :
-- This module contains the EVMDECODE functions from the EVM CPLD Abel files.
--
-- This module provides the DSP EMIF memory decode to control async
-- memory accesses to the CE1 memory space. The devices in the CE1
-- memory space include the CPLD registers, the PCI Controller,
-- the FMIC MVIP Controller, the T1/E1 Transceiver and the expansion
-- memory (daughterboard). Decode logic controls the data bus
-- transceivers that connect the DSP’s EMIF data bus to the periphial,
-- PCI Add–On and external daughterboard data busses. The asynchronous
-- memory control strobes from the DSP are synchronized for use by the
-- PCI controller state machine. Async memory access ready (RDY) generation
-- is implemented for each type device access in the CE1 memory space.
--
-------------------------------------------------------------------------
--   Modification History :
```

```
--
-- Revision: 0
-- Date:     04/02/98
-- Author:      Don Curry (DNA)
-- Description: Initial conversion from ABEL version of EVM CPLD.
--
-- Revision: 1
-- Date:     05/04/98
-- Author:      Don Curry (DNA)
-- Description:
--
-- Added McEVM specific signals and changed emif_req to be sync'd
-- to PCI clock.  This prevents the possibility that an address
-- transition during a rising edge of PCI clock will cause a
-- erroneous emif request to the PCI state machine.
--
--------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY decode IS
   PORT(
-- active low signals are indicated with '_l' appended to signal name.
   ------------------------------------------------------------------------
   -- INPUTS
   ------------------------------------------------------------------------
   brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
   pci_det_l : IN STD_LOGIC ; -- PCI detection indicator
```

```
pciclk    : IN STD_LOGIC ; -- Buffered PCI clk (33MHz max)
clk_a     : IN STD_LOGIC ; -- Osc. A (33.25MHz)
ce1_l     : IN STD_LOGIC ; -- EMIF CE1 memory space enable
ce2_l     : IN STD_LOGIC ; -- EMIF CE2 memory space enable
ce3_l     : IN STD_LOGIC ; -- EMIF CE3 memory space enable
are_l     : IN STD_LOGIC ; -- EMIF async memory read strobe
awe_l     : IN STD_LOGIC ; -- EMIF async memory write strobe
ea_hi     : IN STD_LOGIC_VECTOR(5 DOWNTO 0) ;
                 -- EMIF Address bits 21:16
ao_bsy    : IN STD_LOGIC ; -- Add-On S.M. busy
emif_ack  : IN STD_LOGIC ; -- Add-On S.M. EMIF access ack.
pb_ack    : IN STD_LOGIC ; -- Peripherial bus access ack.
xrdy      : IN STD_LOGIC ; -- EMIF async acc. ready from DB
ce2sden      : IN STD_LOGIC ; -- CE2 SDRAM enable from registers
ce3sden      : IN STD_LOGIC ; -- CE3 SDRAM enable from registers
-----------------------------------------------------------------------
-- OUTPUTS
-----------------------------------------------------------------------
dsp2aod_l : OUT STD_LOGIC ;   -- Enables GD(31:0) to AOD(31:0)
dsp2gd_l  : OUT STD_LOGIC ;   -- Enables ED(31:0) to GD(31:0)
dsp2xd_l  : OUT STD_LOGIC ;   -- Enables GD(31:0) to XD(31:0)
awe_pci_l : OUT STD_LOGIC ;   -- awe_l synced to pci clock
are_pci_l : OUT STD_LOGIC ;   -- are_l synced to pci clock
awe_osca_l   : OUT STD_LOGIC ;   -- awe_l synced to clka clock
are_osca_l   : OUT STD_LOGIC ;   -- are_l synced to clka clock
fmic_req  : OUT STD_LOGIC ;   -- Synchronized FMIC decode
falc_req  : OUT STD_LOGIC ;   -- Synchronized FALC decode
emif_req  : OUT STD_LOGIC ;   -- Synchronized EMIF decode
cpld_cs      : OUT STD_LOGIC ;   -- CPLD DSP register chip select
```

```
   ardy      : OUT STD_LOGIC  –– EMIF async access ready
   );
END decode ;
ARCHITECTURE rtl OF decode IS
–– Define types used
–– Define constants for readability
  CONSTANT true_h   : STD_LOGIC := ’1’ ;
  CONSTANT false_h  : STD_LOGIC := ’0’ ;
  CONSTANT true_l   : STD_LOGIC := ’0’ ;
  CONSTANT false_l  : STD_LOGIC := ’1’ ;
  –– EMIF CE1 space address definitions
  CONSTANT pci_reg_addr: STD_LOGIC_VECTOR(5 DOWNTO 0) := ”110000” ; –– 0x0130/0x0170
  CONSTANT pci_fifo_addr: STD_LOGIC_VECTOR(5 DOWNTO 0) := ”110001” ; –– 0x0131/0x0171
  CONSTANT fmic_addr   : STD_LOGIC_VECTOR(5 DOWNTO 0) := ”110100” ; –– 0x0134/0x0174
  CONSTANT falc_addr   : STD_LOGIC_VECTOR(5 DOWNTO 0) := ”110101” ; –– 0x0135/0x0175
  CONSTANT dsp_reg_addr: STD_LOGIC_VECTOR(5 DOWNTO 0) := ”111000” ; –– 0x0138/0x0178
–– Internal signal declarations
  SIGNAL pcicntlr_rdy  : STD_LOGIC ;–– PCI controller EMIF access ready
  SIGNAL dspreg_rdy : STD_LOGIC ;–– DSP register EMIF access ready
  SIGNAL pb_rdy     : STD_LOGIC ;–– Peripherial bus (FALC/FMIC) access ready
  SIGNAL default_rdy   : STD_LOGIC ;–– default async ready
  SIGNAL ext_dta_l  : STD_LOGIC ;–– external data bus enable
  SIGNAL emif_rq : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL fmic_rq : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL falc_rq : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL emif_request  : STD_LOGIC ;
  SIGNAL fmic_request  : STD_LOGIC ;
  SIGNAL falc_request  : STD_LOGIC ;
  SIGNAL rd_or_wr_l : STD_LOGIC ;
  SIGNAL pci_rdy : STD_LOGIC ;
  SIGNAL rdy_clr_l  : STD_LOGIC ;

BEGIN
-- Pass internal siganls to port outputs
  dsp2xd_l <= ext_dta_l ;
  emif_req <= emif_request ;
  fmic_req <= fmic_request ;
  falc_req <= falc_request ;

-- Concurrent statements
  -- make sure request are valid for at least one clock prior to sending
  -- request to state machine
  emif_request <= true_h WHEN (emif_rq = ”11”) ELSE
                  false_h ;
  fmic_request <= true_h WHEN (fmic_rq = ”11”) ELSE
                  false_h ;
  falc_request <= true_h WHEN (falc_rq = ”11”) ELSE
                  false_h ;

  rd_or_wr_l <= true_l WHEN ((are_l = true_l) OR
                             (awe_l = true_l)) ELSE
                false_l ;
  -- data bus transceiver control
  -- CE1 and read or write and specific address and S.M. not busy
  dsp2aod_l <= true_l WHEN ((ce1_l = true_l) AND
              (rd_or_wr_l = true_l) AND
              (ao_bsy = false_h) AND
              ((ea_hi = pci_reg_addr) OR
             (ea_hi = pci_fifo_addr))) ELSE
        false_l ;
-- CE1 or CE2/CE3 and cd2sden/cd3sden disabled
dsp2gd_l <= true_l WHEN (((ce1_l = true_l) AND
            (rd_or_wr_l = true_l))
          OR
           ((ce2_l = true_l) AND
            (ce2sden = false_h))
          OR
           ((ce3_l = true_l) AND
            (ce3sden = false_h))) ELSE
        false_l ;
-- CE1 address 0xXX0XXXXX – 0xXX2XXXXX, CE2/CE3 and cd2sden/cd3sden disabled
ext_dta_l <= true_l WHEN (((ce1_l = true_l) AND
             (rd_or_wr_l = true_l) AND
                           (ea_hi(5 DOWNTO 4) < ”11”))
                          OR
             ((ce2_l = true_l) AND
              (rd_or_wr_l = true_l) AND
              (ce2sden = false_h))
            OR
             ((ce3_l = true_l) AND
              (rd_or_wr_l = true_l) AND
              (ce3sden = false_h))) ELSE
       false_l ;
-- DSP register access
cpld_cs <= true_h WHEN ((ce1_l = true_l) AND
                        (ea_hi = dsp_reg_addr)) ELSE
      false_h ;
```

```
-- PCI controller EMIF access ready generation
pcicntlr_rdy <= true_h WHEN ((pci_det_l = false_l) OR
               (pci_rdy = true_h)) ELSE
       false_h ;
-- DSP register EMIF access ready generation
dspreg_rdy <= true_h WHEN (rd_or_wr_l = true_l) ELSE
     false_h ;
-- default async ready generation
default_rdy <= true_h WHEN ((ce1_l = true_l) OR
              ((ce1_l = false_l) AND
               (rd_or_wr_l = true_l))) ELSE
      false_h ;
-- Async READY mux
ardy <= pcicntlr_rdy WHEN ((ce1_l = true_l) AND
             ((ea_hi = pci_reg_addr) OR
              (ea_hi = pci_fifo_addr))) ELSE
   dspreg_rdy   WHEN ((ce1_l = true_l) AND
             (ea_hi = dsp_reg_addr)) ELSE
   pb_rdy       WHEN ((ce1_l = true_l) AND
             ((ea_hi = falc_addr) OR
              (ea_hi = fmic_addr))) ELSE
   xrdy         WHEN (((ce1_l = true_l) AND
                           (ea_hi(5 DOWNTO 4) < ”11”))
                          OR
             ((ce2_l = true_l) AND
              (ce2sden = false_h))
            OR
             ((ce3_l = true_l) AND
              (ce3sden = false_h))) ELSE
```

```
        default_rdy ;
    -- clear for ready signals
    rdy_clr_l <= true_l WHEN ((brd_rst_l = true_l) OR
                              (rd_or_wr_l = false_l)) ELSE
                 false_l ;

  -- Sequential statements
  -- generate emif ready by using emif_ack as clock and async reset with
  -- both are and awe high.
  e_rdy:PROCESS(emif_ack, rdy_clr_l)
  BEGIN
    IF (rdy_clr_l = true_l) THEN
      pci_rdy <= false_h ;
    ELSIF ((emif_ack'EVENT) AND
           (emif_ack = true_h)) THEN
      pci_rdy <= true_h ;
    END IF ; -- clocked IF
  END PROCESS e_rdy ;
  -- generate emif ready by using emif_ack as clock and async reset with
  -- both are and awe high.
  p_rdy:PROCESS(pb_ack, rdy_clr_l)
  BEGIN
    IF (rdy_clr_l = true_l) THEN
      pb_rdy <= false_h ;
    ELSIF ((pb_ack'EVENT) AND
           (pb_ack = true_h)) THEN
      pb_rdy <= true_h ;
    END IF ; -- clocked IF
  END PROCESS p_rdy ;
```

```
-- synchronize decodes to approriate clock.
-- Since ce1 and address signals are async to clocks, must ensure that
-- clock did not capture transition of address and give a false request.
-- To do this address must be valid for two clocks prior to request
-- being issued.
pci_sync:PROCESS(pciclk, brd_rst_l)
BEGIN
  IF (brd_rst_l = true_l) THEN
    emif_rq <= ”00” ;
    are_pci_l <= false_l ;
    awe_pci_l <= false_l ;
  ELSIF ((pciclk’EVENT) AND
         (pciclk = true_h)) THEN
    –- requires that DSP’s hold + setup > 1 PCI clock
    are_pci_l <= are_l ;
    awe_pci_l <= awe_l ;
    IF (emif_ack = false_h) THEN
      emif_rq(1) <= emif_rq(0) ;
      IF ((ce1_l = true_l) AND
          ((ea_hi = pci_reg_addr) OR
           (ea_hi = pci_fifo_addr))) THEN
        emif_rq(0) <= true_h ;
      ELSE
        emif_rq(0) <= false_h ;
      END IF ;
    ELSE
      emif_rq <= ”00” ;
    END IF ;
  END IF ; –- clocked IF
END PROCESS pci_sync ;

osca_sync:PROCESS(clk_a, brd_rst_l)
BEGIN
  IF (brd_rst_l = true_l) THEN
    falc_rq <= ”00” ;
    fmic_rq <= ”00” ;
    are_osca_l <= false_l ;
    awe_osca_l <= false_l ;
  ELSIF ((clk_a’EVENT) AND
         (clk_a = true_h)) THEN

    are_osca_l <= are_l ;
    awe_osca_l <= awe_l ;
    IF (pb_ack = false_h) THEN
      fmic_rq(1) <= fmic_rq(0) ;
    ELSE
      fmic_rq(1) <= false_h ;
    END IF ;

    IF ((pb_ack = false_h) AND
        (ce1_l = true_l) AND
        (ea_hi = fmic_addr)) THEN
      fmic_rq(0) <= true_h ;
    ELSE
      fmic_rq(0) <= false_h ;
    END IF ;
    IF (pb_ack = false_h) THEN
      falc_rq(1) <= falc_rq(0) ;
    ELSE
      falc_rq(1) <= false_h ;
    END IF ;

    IF ((pb_ack = false_h) AND
        (ce1_l = true_l) AND
        (ea_hi = falc_addr)) THEN
      falc_rq(0) <= true_h ;
    ELSE
      falc_rq(0) <= false_h ;
    END IF ;
  END IF ; -- clocked IF
END PROCESS osca_sync ;

END rtl ;
```

```
--------------------------------------------------------------------------
--                                    Design For:
--                           Texas Instruments Incorporated
--
--                                    Design By:
--                DNA Enterprises Inc
--                269 West Renner Parkway
--                Richardson, TX 75080
--
--   File name     :  irq_ctrl.vhd
--   Title         :  McEVM Interrupt Controller
--   Module        :  Top–Level McEVM Control
--   Description   :
--  This module contains the EVMINT functions from the EVM CPLD Abel files.
--
--  This module controls the interrupts to the host (via the PCI
--  Controller) and the ’C6201 DSP. The DSP host interrupt and the
--  JTAG TBC can be used to interrupt the host. This modules implements
--  falling edge detectors to determine when these source interrupts
--  occur to force a PCI mailbox interrupt in the S5933 PCI Controller
--  which causes and INTA# host interrupt. Logic is included to
--  implement a DSP NMI interrupt source. The PCI controller add–on
--  IRQ# is passed to the DSP via its EXT_INT4 input. Additionally,
--  PCI bus master read (EXT_INT5) and write (EXT_INT6) interrupts
--  are generated by state machines in this module that monitor the PCI
--  controllers FIFO FULL and EMPTY flags to determine when data can
--  be transferred between the PCI Controller and DSP memory space.
--  These interrupts can be used by the DSP to trigger ISR’s or more
-- typically used to trigger background DMA transfers. Optionally,
-- the DSP could even poll these interrupts or the FIFO flags themselves
-- to control data transfers.
--
-- The DSP can also be interrupted via the T1/E1 transceiver or the
-- daughterboard.  This module takes in the two interrupt sources
-- (both active high, level interrupts) and generate a rising edge
-- interrupt to the DSP (EXT_INT7).  To clear the interrupts the DSP
-- must write to the Interrupt control register with the appropriate
-- bits set.  If another interrupt is pending, another edge will be
-- generated so that the interrupt can be serviced.  The clear bits
-- can also be used to mask interrupts from a particular source.  If
-- the clear bit is set no interrupts from that source are latched
-- and thus cannot cause and interrupt to the DSP.
--
-------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     04/02/98
-- Author:      Don Curry (DNA)
-- Description:  Initial conversion from ABEL version of EVM CPLD.
--
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY irq_ctrl IS
```

```
   PORT(
-- active low signals are indicated with ’_l’ appended to signal name.
   -----------------------------------------------------------------------
   -- INPUTS
   -----------------------------------------------------------------------
   brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
   dsp_hint_l    : IN STD_LOGIC ; -- DSP HPI interrupt
   tbc_int_l : IN STD_LOGIC ; -- TBC host interrupt
   pciclk    : IN STD_LOGIC ; -- Buffered PCI clk (33MHz max)
   pci_det_l : IN STD_LOGIC ; -- PCI detection indicator
   pci_irq_l : IN STD_LOGIC ; -- Interrupt from add–on device
   rdempty       : IN STD_LOGIC ; -- PCI read FIFO empty flag
   wrfull    : IN STD_LOGIC ; -- PCI write FIFO full flag
   hinten    : IN STD_LOGIC ; -- HPI host interrupt enable
   tbcinten  : IN STD_LOGIC ; -- TBC host interrupt enable
   dspnmi    : IN STD_LOGIC ; -- DSP NMI interrupt from host
   nmisel    : IN STD_LOGIC ; -- NMI source (host/xcvr)
   nmien     : IN STD_LOGIC ; -- NMI enable
   pcimren       : IN STD_LOGIC ; -- PCI master read IRQ enable
   pcimwen       : IN STD_LOGIC ; -- PCI master write IRQ enable
   rdfifo_l  : IN STD_LOGIC ; -- PCI FIFO read strobe
   wrfifo_l  : IN STD_LOGIC ; -- PCI FIFO write strobe
   clk_a     : IN STD_LOGIC ; -- Osc A
   clr_falcint   : IN STD_LOGIC ; -- T1/E1 xcvr interrupt
   clr_dbint : IN STD_LOGIC ; -- Daughterboard interrupt
   falc_int  : IN STD_LOGIC ; -- T1/E1 xcvr interrupt
   db_int    : IN STD_LOGIC ; -- Daughterboard interrupt
   -----------------------------------------------------------------------
   -- OUTPUTS
   -----------------------------------------------------------------------
   db_falc_int  : OUT STD_LOGIC ;   -- DB/FALC interrupt (EXT_INT7)
   ltchd_falc_int   : OUT STD_LOGIC ;   -- Latched FALC interrupt
   ltchd_db_int : OUT STD_LOGIC ;   -- Latched DB interrupt
   pc_int    : OUT STD_LOGIC ;   -- Add-on interrupt to PCI cntlr
   pci_int       : OUT STD_LOGIC ;   -- PCI to DSP interrupt (EXT_INT4)
   nmi    : OUT STD_LOGIC ;   -- NMI to DSP
   pcimrd_int    : OUT STD_LOGIC ;   -- PCI mstr read IRQ to DSP (EXT_INT5)
   pcimwr_int    : OUT STD_LOGIC  -- PCI mstr write IRQ to DSP (EXT_INT6)
   );
END irq_ctrl ;
ARCHITECTURE rtl OF irq_ctrl IS
-- Define types used
-- Define constants for readability
  CONSTANT true_h   : STD_LOGIC := ’1’ ;
  CONSTANT false_h  : STD_LOGIC := ’0’ ;
  CONSTANT true_l   : STD_LOGIC := ’0’ ;
  CONSTANT false_l  : STD_LOGIC := ’1’ ;
  -- PCI Master Read State Assignments
  CONSTANT mrd_idle : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”00” ;
  CONSTANT mrd_int  : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”01” ;
  CONSTANT mrd_wait : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”10” ;
  CONSTANT mrd_done : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”11” ;
  -- PCI Master Write State Assignments
  CONSTANT mwr_idle : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”00” ;
  CONSTANT mwr_int  : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”01” ;
  CONSTANT mwr_wait : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”10” ;
  CONSTANT mwr_done : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”11” ;
-- Internal signal declarations
```

```
  -- falling edge detector signals
  SIGNAL hint_db1_l : STD_LOGIC ;
  SIGNAL hint_db2_l : STD_LOGIC ;
  SIGNAL tbcint_db1_l  : STD_LOGIC ;
  SIGNAL tbcint_db2_l  : STD_LOGIC ;
  -- Master read/write state signals
  SIGNAL mrd     : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL mwr     : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  -- Master read/write next state signals
  SIGNAL mrd_next_state : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  SIGNAL mwr_next_state : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  -- clear interrupt edge detect
  SIGNAL ltchd_dbint   : STD_LOGIC ;
  SIGNAL ltchd_falcint : STD_LOGIC ;
BEGIN
-- Pass internal siganls to port outputs
  ltchd_db_int <= ltchd_dbint ;
  ltchd_falc_int <= ltchd_falcint ;

-- Concurrent statements
  -- Select NMI source and pass through if enabled
  -- McEVM nmisel tied low
  nmi <= dspnmi   WHEN ((nmien = true_h) AND (nmisel = false_h)) ELSE
    false_h ;
  -- PCI interrupt only active if PCI detected and not in reset
  pci_int <= true_h WHEN ((pci_det_l = true_l) AND
           (brd_rst_l = false_l) AND
           (pci_irq_l = true_l)) ELSE
        false_h ;
  -- DSP interrupt is sourced from FALC and/or the daughterboard
  -- Both interrupts are assumed to be active high level interrupts.
  db_falc_int <= true_h WHEN (((ltchd_dbint = true_h) OR
                               (ltchd_falcint = true_h))
                              AND
                              ((clr_dbint = false_h) AND
                               (clr_falcint = false_h))) ELSE
                false_h ;

-- Sequential statements
-- PCI Master Read State Machine Next State decode
mrd_sm:PROCESS(mrd, pcimren, rdempty, rdfifo_l)
BEGIN
  CASE mrd IS
    WHEN mrd_idle =>
      -- Wait for master read to be enabled and FIFO not empty
      IF ((pcimren = true_h) AND (rdempty = false_h)) THEN
   mrd_next_state <= mrd_int ;
      ELSE
   mrd_next_state <= mrd_idle ;
      END IF ;
    WHEN mrd_int =>
      -- Master read is enabled and FIFO is not empty
      -- Always go next state
      mrd_next_state <= mrd_wait ;
    WHEN mrd_wait =>
      -- Wait for DSP to read FIFO or DSP to disable master read
      IF ((pcimren = false_h) OR
    (rdfifo_l = true_l)) THEN
   mrd_next_state <= mrd_done ;
      ELSE
   mrd_next_state <= mrd_wait ;
      END IF ;
    WHEN mrd_done =>
      -- Wait for DSP to complete FIFO read or DSP to disable master read
      IF ((pcimren = false_h) OR
    (rdfifo_l = false_l)) THEN
   mrd_next_state <= mrd_idle ;
      ELSE
   mrd_next_state <= mrd_done ;
      END IF ;
    WHEN OTHERS => mrd_next_state <= mrd_idle ;
  END CASE ;
END PROCESS mrd_sm ;
-- PCI Master Write State Machine Next State decode
mwr_sm:PROCESS(mwr, pcimwen, wrfull, wrfifo_l)
BEGIN
  CASE mwr IS
    WHEN mwr_idle =>
      -- Wait for master write to be enabled and FIFO not full
      IF ((pcimwen = true_h) AND (wrfull = false_h)) THEN
   mwr_next_state <= mwr_int ;
      ELSE
   mwr_next_state <= mwr_idle ;
      END IF ;
    WHEN mwr_int =>
      -- Master write is enabled and FIFO is not full
      -- Always go next state
      mwr_next_state <= mwr_wait ;
    WHEN mwr_wait =>
      -- Wait for DSP to read FIFO or DSP to disable master write
      IF ((pcimwen = false_h) OR
    (wrfifo_l = true_l)) THEN
   mwr_next_state <= mwr_done ;
      ELSE
   mwr_next_state <= mwr_wait ;
      END IF ;
    WHEN mwr_done =>
      -- Wait for DSP to complete FIFO read or DSP to disable master write
      IF ((pcimwen = false_h) OR
    (wrfifo_l = false_l)) THEN
   mwr_next_state <= mwr_idle ;
      ELSE
   mwr_next_state <= mwr_done ;
      END IF ;
    WHEN OTHERS => mwr_next_state <= mwr_idle ;
  END CASE ;
END PROCESS mwr_sm ;
pci_seq:PROCESS(pciclk, brd_rst_l)
-- synchronize read, write and emif request to pciclk
BEGIN
  IF (brd_rst_l = true_l) THEN
    hint_db1_l <= false_l ;
    hint_db2_l <= false_l ;
    tbcint_db1_l <= false_l ;
    tbcint_db2_l <= false_l ;
    pc_int <= false_h ;
    pcimrd_int <= false_h ;
    pcimwr_int <= false_h ;
    mrd <= mrd_idle ;
    mwr <= mwr_idle ;
  ELSIF ((pciclk'EVENT) AND
     (pciclk = true_h)) THEN

    -- PC interrupt rising edge whenever DSP host interrupt or TBC interrupt
    -- have a falling edge
    IF (((hinten = true_h) AND
       (hint_db1_l = true_l) AND (hint_db2_l = false_l)) OR
       ((tbcinten = true_h) AND
       (tbcint_db1_l = true_l) AND (tbcint_db2_l = false_l))) THEN
      pc_int <= true_h ;
    ELSE
      pc_int <= false_h ;
    END IF ;
    -- take external interrupts into falling edge detector shift registers
    hint_db1_l <= dsp_hint_l ;
    hint_db2_l <= hint_db1_l ;
    tbcint_db1_l <= tbc_int_l ;
    tbcint_db2_l <= tbcint_db1_l ;
    IF (pci_det_l = true_l) THEN
      -- clock next state values into state machine only if PCI is detected
      mrd <= mrd_next_state ;
      mwr <= mwr_next_state ;
      -- Assert DSP EXT_INT5 when going into mrd_int state
      -- De-assert DSP EXT_INT5 when going into mrd_idle state
      IF (mrd_next_state /= mrd_idle) THEN
    pcimrd_int <= true_h ;
       ELSE
    pcimrd_int <= false_h ;
       END IF ;
       -- Assert DSP EXT_INT6 when going into mwr_int state
       -- De-assert DSP EXT_INT6 when going into mwr_idle state
       IF (mwr_next_state /= mwr_idle) THEN
    pcimwr_int <= true_h ;
       ELSE
    pcimwr_int <= false_h ;
       END IF ;
     END IF ; -- pci_det_l IF
   END IF ; -- clocked IF
 END PROCESS pci_seq ;

 dsp_seq:PROCESS(clk_a, brd_rst_l)
 -- generate interrupt clears ind sync'd interrupts
 BEGIN
   IF (brd_rst_l = true_l) THEN
     ltchd_dbint <= false_h ;
     ltchd_falcint <= false_h ;
   ELSIF ((clk_a'EVENT) AND
          (clk_a = true_h)) THEN
     IF (clr_dbint = false_h) THEN
       IF (ltchd_dbint = false_h) THEN
         ltchd_dbint <= db_int ;
       END IF ;
     ELSE
       ltchd_dbint <= false_h ;
```

```
  END IF ;

  IF (clr_falcint = false_h) THEN
   IF (ltchd_falcint = false_h) THEN
    ltchd_falcint <= falc_int ;
   END IF ;
  ELSE
   ltchd_falcint <= false_h ;
  END IF ;

 END IF ; -- clocked IF
END PROCESS dsp_seq ;

END rtl ;
```

```
-------------------------------------------------------------------------
--                                     Design For:
--                          Texas Instruments Incorporated
--
--                                     Design By:
--              DNA Enterprises Inc
--              269 West Renner Parkway
--              Richardson, TX 75080
--
--   File name      :  pci_cntlr.vhd
--   Title          :  McEVM PCI Interface Controller
--   Module         :  Top-Level McEVM Control
--   Description    :
-- This module contains the EVMPCI functions from the EVM CPLD Abel files.
--
-- This module controls the slave (target) interface between the PCI
-- Controller and the JTAG TBC, DSP HPI and CPLD PCI registers. It
-- also controls the interface for DSP EMIF accesses to the PCI
-- controller which include PCI Bus Master read/write transfers.
-- This module provides the state machine(s) necessary to control
-- the signals that are required to transfer data between the PCI
-- controller and the DSP. The CPLD only interfaces to the PCI
-- controllers add-on bus interface and not the actual PCI bus.
--
-------------------------------------------------------------------------
--   Modification History :
--
-- Revision: 0
-- Date:     04/29/98
-- Author:      Don Curry (DNA)
-- Description: Initial conversion from ABEL version of EVM CPLD.
--
-- Revision: 1
-- Date:     05/01/98
-- Author:      Don Curry (DNA)
-- Description: Re-wrote entire state machine due to ensure what
--        I did would work as I thought without any surprises.
--
-- Revision: 2
-- Date:     05/05/98
-- Author:      Don Curry (DNA)
-- Description: Added state to HPI accesses during x1 mode to
--        assure HDS1 pulse width requirements.
--
-- Revision: 3
-- Date:     08/07/98
-- Author:      Don Curry (DNA)
-- Description: Changed add-on address from pci_aptd_adr to ea_lo
--              Added multiplex feature for PTADR
--              Found tbca_hpi problem -- bit vector control shifted right by one.  Fixed WFD
--  Revision:   4
--  Date:       08/10/98
--  Author:     Bill Dempsey (DNA)
--  Description:    Changed C6x write access to AMCC from 2 clk wide pulse to 1 clk wide pulse
--
--  Revision:   5
--  Date:       08/14/98
--  Author:      Bill Dempsey (DNA)
--  Description:   Found that back-to-back PTNUM=2 accesses without deassertion of PTATN caused state
--                 machine to restart without asserting AMCC chip select ’ao_sel_l’
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY pci_control IS
   PORT(
-- active low signals are indicated with ’_l’ appended to signal name.
   -----------------------------------------------------------------------
   -- INPUTS
   -----------------------------------------------------------------------
   brd_rst_l : IN STD_LOGIC ; -- Reset from volt. super.
   pciclk    : IN STD_LOGIC ; -- Buffered PCI clk (33MHz max)
   clk_mode  : IN STD_LOGIC ; -- DSP clock mode
   pci_det_l : IN STD_LOGIC ; -- PCI detection indicator
   ptatn_l      : IN STD_LOGIC ; -- Pass-thru attention signal
   ptburst_l : IN STD_LOGIC ; -- Pass-thru burst signal
   ptnum     : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                    -- Pass-thru region number
   ptwr      : IN STD_LOGIC ; -- Pass-thru access type (R=0/W=1)
   pt_be_l      : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                    -- Latched pass-thru byte enables
   pt_addr      : IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                    -- Latched pass-thru address
   are_pci_l : IN STD_LOGIC ; -- are_l synced to pciclk
   awe_pci_l : IN STD_LOGIC ; -- awe_l synced to pciclk
```

```
be_l      : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- EMIF byte enables
ea16      : IN STD_LOGIC ; -- EMIF address bit 16
ea_lo     : IN STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                 -- EMIF address bits 6:2
tbc_rdy_l : IN STD_LOGIC ; -- JTAG TBC Ready
dsp_hrdy_l   : IN STD_LOGIC ; -- DSP HPI Ready
emif_req  : IN STD_LOGIC ; -- Synchronized EMIF access request
------------------------------------------------------------------------
-- OUTPUTS
------------------------------------------------------------------------
pci_flt_l : OUT STD_LOGIC ;   -- Tri-state PCI controller outputs
pt_adr_l  : OUT STD_LOGIC ;   -- Pass-thru address request
pt_rdy_l  : OUT STD_LOGIC ;   -- Pass-thru ready signal
ao_sel_l  : OUT STD_LOGIC ;   -- Add-on register access
ao_wr_l      : OUT STD_LOGIC ;   -- Add-on write strobe
ao_rd_l      : OUT STD_LOGIC ;   -- Add-on read strobe
ao_adr    : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                 -- Add-on address bits 6:2
ao_be_l      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 -- Add-on byte enables
rdfifo_l  : OUT STD_LOGIC ;   -- Read FIFO strobe
wrfifo_l  : OUT STD_LOGIC ;   -- Write FIFO strobe
pcireg_oe : OUT STD_LOGIC ;   -- PCI register output enable
pcireg_ce : OUT STD_LOGIC ;   -- PCI register clock enable
tbc_wr_l  : OUT STD_LOGIC ;   -- TBC write strobe
tbc_rd_l  : OUT STD_LOGIC ;   -- TBC read strobe
hcs_l     : OUT STD_LOGIC ;   -- DSP HPI chip select
hds1_l    : OUT STD_LOGIC ;   -- DSP HPI data strobe
```

```
      hrw    : OUT STD_LOGIC ;   -- DSP HPI read/write control
      tbca_hpic : OUT STD_LOGIC_VECTOR(4 DOWNTO 0) ;
                       -- TBC address bits 4:0/HPI control
      ao_bsy    : OUT STD_LOGIC ;   -- Add-on S.M. busy signal
      emif_ack  : OUT STD_LOGIC  -- EMIF access acknowledge
      );
END pci_control ;
ARCHITECTURE rtl OF pci_control IS
-- Define types used
  TYPE ao_state_type IS (idle, ao_idle, pt_idle,
                         ao_s1, ao_s2, ao_s3,
                         pt_s1, pt_s2, pt_s3,
                         pt_hpi_s1, pt_hpi_s2, pt_hpi_s3, pt_hpi_s3a,
                         pt_hpi_s4, pt_hpi_s5, pt_hpi_s5a, pt_hpi_s6,
                         pt_hpi_s7,
                         done) ;
-- Define constants for readability
  CONSTANT true_h   : STD_LOGIC := ’1’ ;
  CONSTANT false_h  : STD_LOGIC := ’0’ ;
  CONSTANT true_l   : STD_LOGIC := ’0’ ;
  CONSTANT false_l  : STD_LOGIC := ’1’ ;
  CONSTANT tbc_acc  : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”00” ;
  CONSTANT reg_acc  : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”01” ;
  CONSTANT hpi_acc  : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”10” ;
  CONSTANT hpib_acc : STD_LOGIC_VECTOR(1 DOWNTO 0) := ”11” ;
  CONSTANT pci_aptd_adr : STD_LOGIC_VECTOR(4 DOWNTO 0) := ”01011” ;
-- Internal signal declarations
  SIGNAL ao_state   : ao_state_type ;
  SIGNAL ao_next_state  : ao_state_type ;
  SIGNAL tbcrdy     : STD_LOGIC ;-- Synchronized & inverted TBC ready
  SIGNAL hrdy    : STD_LOGIC ;-- Synchronized & inverted HPI ready
  SIGNAL hcntl1     : STD_LOGIC ;-- Host control bit 1
  SIGNAL hcntl0     : STD_LOGIC ;-- Host control bit 0
  SIGNAL tbc_hpi_ctrl  : STD_LOGIC_VECTOR(8 DOWNTO 0) ;
               -- TBC and HPI Control vector
  SIGNAL ao_ctrl : STD_LOGIC_VECTOR(11 DOWNTO 0) ;
               -- Add-on bus Control vector
  SIGNAL pt_ctrl : STD_LOGIC_VECTOR(5 DOWNTO 0) ;
               -- Pass-thru Control vector
-- rev 3
  SIGNAL adr_mux_sel: STD_LOGIC ;-- mux select between ao_adr and ea_lo
  SIGNAL ea_lo_clkd : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
-- Pass internal siganls to port outputs
-- Concurrent statements
  -- tri-state PCI controllers output when not in slot
  pci_flt_l <= NOT pci_det_l ;
  ao_adr <= ea_lo_clkd WHEN (adr_mux_sel = true_h) ELSE
            pci_aptd_adr ;

-- Sequential statements
ADR_REG: PROCESS(adr_mux_sel)
BEGIN
    IF (adr_mux_sel'EVENT) AND
     (adr_mux_sel = true_h) THEN
       ea_lo_clkd <= ea_lo;
    END IF;
END PROCESS ADR_REG;
```

```
-- PCI Master Read State Machine Next State decode
ao_sm:PROCESS(ao_state, emif_req, awe_pci_l, are_pci_l,
              ptnum, ptatn_l, hrdy, ptburst_l, clk_mode)
BEGIN
  CASE ao_state IS
  ------------------------------------------------------------------------
    WHEN idle =>
  ------------------------------------------------------------------------
      IF (emif_req = true_h) THEN
        ao_next_state <= ao_idle ;
      ELSIF (ptatn_l = true_l) THEN
        ao_next_state <= pt_idle ;
      ELSE
        ao_next_state <= ao_state ;
      END IF ;
  ------------------------------------------------------------------------
    WHEN ao_idle => -- wait for sync'd read or write strobe
  ------------------------------------------------------------------------
      IF ((awe_pci_l = true_l) OR
          (are_pci_l = true_l)) THEN
        ao_next_state <= ao_s1 ;
      ELSE
        ao_next_state <= ao_state ;
      END IF ;
  ------------------------------------------------------------------------
    WHEN pt_idle =>
  ------------------------------------------------------------------------
      CASE ptnum IS
        WHEN tbc_acc => ao_next_state <= pt_s1 ;
      WHEN reg_acc => ao_next_state <= pt_s1 ;
      WHEN hpi_acc => ao_next_state <= pt_hpi_s1 ;
      WHEN hpib_acc => ao_next_state <= pt_hpi_s1 ;
      WHEN OTHERS => ao_next_state <= ao_state ;
    END CASE ;
------------------------------------------------------------------------
  WHEN ao_s1 =>
------------------------------------------------------------------------
    ao_next_state <= ao_s2 ;
------------------------------------------------------------------------
  WHEN ao_s2 =>
------------------------------------------------------------------------
    ao_next_state <= ao_s3 ;
------------------------------------------------------------------------
  WHEN ao_s3 =>
------------------------------------------------------------------------
    IF ((awe_pci_l = false_l) AND
        (are_pci_l = false_l)) THEN
      ao_next_state <= done ;
    ELSE
      ao_next_state <= ao_state ;
    END IF ;
------------------------------------------------------------------------
  WHEN pt_s1 =>
------------------------------------------------------------------------
    ao_next_state <= pt_s2 ;
------------------------------------------------------------------------
  WHEN pt_s2 =>
------------------------------------------------------------------------
```

```
      ao_next_state <= pt_s3 ;
  ------------------------------------------------------------------------
    WHEN pt_s3 =>
  ------------------------------------------------------------------------
      ao_next_state <= done ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s1 =>
  ------------------------------------------------------------------------
      ao_next_state <= pt_hpi_s2 ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s2 =>
  ------------------------------------------------------------------------
--
--  Removed by WFD on 08/08/98.  Found that state machine wasn’t checking HRDY correctly
--
--      IF (clk_mode = true_h) THEN
--        ao_next_state <= pt_hpi_s3 ;
--      ELSE
--        ao_next_state <= pt_hpi_s3a ;
--      END IF ;
      ao_next_state <= pt_hpi_s3a ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s3a =>
  ------------------------------------------------------------------------
      ao_next_state <= pt_hpi_s3 ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s3 =>
  ------------------------------------------------------------------------
      IF (hrdy = true_h) THEN
```

```
        ao_next_state <= pt_hpi_s4 ;
      ELSE
        ao_next_state <= ao_state ;
      END IF ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s4 =>
  ------------------------------------------------------------------------
--
--  Removed by WFD on 08/08/98.  Found that state machine wasn’t checking HRDY correctly
--
--      IF (clk_mode = true_h) THEN
--        ao_next_state <= pt_hpi_s5 ;
--      ELSE
--        ao_next_state <= pt_hpi_s5a ;
--      END IF ;
      ao_next_state <= pt_hpi_s5a ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s5a =>
  ------------------------------------------------------------------------
      ao_next_state <= pt_hpi_s5 ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s5 =>
  ------------------------------------------------------------------------
      IF (hrdy = true_h) THEN
        ao_next_state <= pt_hpi_s6 ;
      ELSE
        ao_next_state <= ao_state ;
      END IF ;
  ------------------------------------------------------------------------
```

```
    WHEN pt_hpi_s6 =>
  ------------------------------------------------------------------------
      ao_next_state <= pt_hpi_s7 ;
  ------------------------------------------------------------------------
    WHEN pt_hpi_s7 =>
  ------------------------------------------------------------------------
--      IF (ptnum = hpi_acc) THEN
--        ao_next_state <= done ;
--  Rev 5 commented out these next three lines
--      IF ((ptatn_l = false_l) AND
--          (ptburst_l = false_l)) THEN
--        ao_next_state <= done ;
--      ELSIF ((ptnum = hpib_acc) AND  -- rev 5
--   ELSEIF (ptatn_l = false_l) AND -- rev 5
--             (ptburst_l = false_l)) THEN
--        ao_next_state <= done ;
--      ELSIF (ptatn_l = true_l) THEN
--(ptnum = hpib_acc)
--        ao_next_state <= pt_idle ;  -- rev 5 (was pt_num_1)
--      ELSE
--        ao_next_state <= ao_state ;
--      END IF ;
      IF ((ptnum = hpi_acc) OR ((ptatn_l = false_l) AND (ptburst_l = false_l))) THEN
        ao_next_state <= done;
      ELSIF (ptatn_l = true_l) THEN
        ao_next_state <= pt_idle;
      ELSE
        ao_next_state <= ao_state;
      END IF;
  ------------------------------------------------------------------------
    WHEN done =>
  ------------------------------------------------------------------------
      ao_next_state <= idle ;
  ------------------------------------------------------------------------
    WHEN OTHERS => ao_next_state <= idle ;
  ------------------------------------------------------------------------
  END CASE ;
END PROCESS ao_sm ;
seq:PROCESS(pciclk, brd_rst_l)
BEGIN
  IF (brd_rst_l = true_l) THEN
    ao_state <= idle ;     -- Beginning of state machine
    hrdy <= false_h ;      -- Not ready
    tbcrdy <= false_h ;    -- Not ready
    pt_adr_l <= false_l ;  -- No direct PCI address register read
    pt_rdy_l <= false_l ;
    ao_sel_l <= false_l ;
--    ao_adr <= ”00000” ; -- rev 3
    adr_mux_sel <= false_h ; -- rev 3
    ao_be_l <= ”1111” ;
    ao_wr_l <= false_l ;
    ao_rd_l <= false_l ;
    emif_ack <= false_h ;  -- No EMIF acknowledge
    rdfifo_l <= false_l ;
    wrfifo_l <= false_l ;
    tbc_wr_l <= false_l ;
    tbc_rd_l <= false_l ;
    pcireg_oe <= false_h ;
  pcireg_ce <= false_h ;
  hcs_l <= false_l ;
  hds1_l <= false_l ;
  hrw <= false_h ;       -- Host R/W strobe
  tbca_hpic <= ”00011” ;
  ao_bsy <= false_h ;    -- State machine not busy
ELSIF ((pciclk’EVENT) AND
   (pciclk = true_h)) THEN

  -- clock next state values into state machine only if PCI is detected
  IF (pci_det_l = true_l) THEN
    ao_state <= ao_next_state ;
  END IF ;
  -- synchronize various signals to pciclk
  hrdy <= NOT dsp_hrdy_l ;
  tbcrdy <= NOT tbc_rdy_l ;

  -- Host read/write is inverted pass–thru r/w strobe
  hrw <= NOT ptwr ;
  -- Define all control signals relative to current state
  -- set default
  adr_mux_sel <= false_h ; -- rev 3
  CASE ao_state IS
  ------------------------------------------------------------------------
    WHEN idle =>
  ------------------------------------------------------------------------
      IF ((emif_req = false_h) AND
          (ptatn_l = true_l)) THEN
        pt_adr_l <= true_l ;
```

```
          ao_bsy <= true_h ;
        END IF ;
    ------------------------------------------------------------------------
      WHEN ao_idle =>
    ------------------------------------------------------------------------
--        ao_adr <= pci_aptd_adr ;  -- rev 3 change
        adr_mux_sel <= true_h ;  -- rev 3
        IF (ea16 = false_h) THEN
          ao_sel_l <= true_l ;
          ao_be_l <= be_l ;
          IF (are_pci_l = true_l) THEN
            ao_rd_l <= true_l ;
          END IF ;
--  Removed by WFD 08/10/98 to remove double clock wide ao_wr_l
--          IF (awe_pci_l = true_l) THEN
--            ao_wr_l <= true_l ;
--          END IF ;
        ELSE
          ao_be_l <= ”0000” ;
          IF (are_pci_l = true_l) THEN
            rdfifo_l <= true_l ;
          END IF ;
          IF (awe_pci_l = true_l) THEN
            wrfifo_l <= true_l ;
          END IF ;
        END IF ;
    ------------------------------------------------------------------------
      WHEN pt_idle =>
    ------------------------------------------------------------------------
```

```
        pt_adr_l <= false_l ;
--        ao_adr <= pci_aptd_adr ; -- rev 3
        ao_sel_l <= true_l ;
        CASE ptnum IS
          WHEN hpi_acc =>ao_be_l <= x”C” ;
          WHEN hpib_acc =>ao_be_l <= x”C” ;
          WHEN OTHERS => ao_be_l <= ”0000” ;
        END CASE ;
        IF (ptwr = false_h) THEN
          CASE ptnum IS
            WHEN tbc_acc => tbc_rd_l <= true_l ;
            WHEN reg_acc => pcireg_oe <= true_h ;
            WHEN OTHERS => NULL ;
          END CASE ;
        ELSE
          ao_rd_l <= true_l ;
        END IF ;
    ------------------------------------------------------------------------
      WHEN ao_s1 =>
    ------------------------------------------------------------------------
        adr_mux_sel <= true_h ;  -- rev 3
        IF (ea16 = false_h) THEN
          IF (awe_pci_l = true_l) THEN
            ao_wr_l <= true_l ;
          END IF ;
        ELSE
          IF (awe_pci_l = true_l) THEN
            wrfifo_l <= true_l ;
          END IF ;
    END IF ;
------------------------------------------------------------------------
  WHEN ao_s2 =>
------------------------------------------------------------------------
    adr_mux_sel <= true_h ;  -- rev 3
    ao_wr_l <= false_l ;
    wrfifo_l <= false_l ;
    emif_ack <= true_h ;
------------------------------------------------------------------------
  WHEN ao_s3 =>
------------------------------------------------------------------------
    adr_mux_sel <= true_h ;  -- rev 3
    IF ((awe_pci_l = false_l) AND
        (are_pci_l = false_l)) THEN
      emif_ack <= false_h ;
    END IF ;
------------------------------------------------------------------------
  WHEN pt_s1 =>
------------------------------------------------------------------------
    tbca_hpic <= pt_addr;
    IF (ptwr = false_h) THEN
      ao_wr_l <= true_l ;
    ELSE
      CASE ptnum IS
        WHEN tbc_acc => tbc_wr_l <= true_l ;
        WHEN reg_acc => pcireg_ce <= true_h ;
        WHEN OTHERS => NULL ;
      END CASE ;
    END IF ;
    ------------------------------------------------------------------------
      WHEN pt_s2 =>
    ------------------------------------------------------------------------
--        ao_wr_l <= false_l ;
        tbc_wr_l <= false_l ;
        pcireg_ce <= false_h ;
    ------------------------------------------------------------------------
      WHEN pt_s3 =>
    ------------------------------------------------------------------------
        ao_wr_l <= false_l ;
        pt_rdy_l <= true_l ;
    ------------------------------------------------------------------------
      WHEN pt_hpi_s1 =>
    ------------------------------------------------------------------------
        hcs_l <= true_l ;
        tbca_hpic(2) <= false_h ;-- (hhwil) first access
        IF (ptnum = hpi_acc) THEN
          IF (pt_addr(1 DOWNTO 0) = ″10″) THEN -- block access to auto incr if we’re in non–auto space
--            tbca_hpic(3 DOWNTO 2 ) <= ″10″ ; -- hcntl(1:0)
            tbca_hpic(4 DOWNTO 3 ) <= ″11″ ;   -- hcntl(1:0)  Fix: WFD
          ELSE
--            tbca_hpic(3 DOWNTO 3 ) <= pt_addr(1 DOWNTO 0) ;  -- hcntl(1:0)  Fix: WFD
            tbca_hpic(4 DOWNTO 3 ) <= pt_addr(1 DOWNTO 0) ; -- hcntl(1:0)
          END IF ;
        ELSIF (ptnum = hpib_acc) THEN      -- only access auto incr data
--          tbca_hpic(3 DOWNTO 2 ) <= ″10″ ;   -- hcntl(1:0)  Fix: WFD
          tbca_hpic(4 DOWNTO 3 ) <= ″10″ ; -- hcntl(1:0)
        END IF ;
        IF (ptwr = true_h) THEN
      tbca_hpic(1 DOWNTO 0) <= pt_be_l(1 DOWNTO 0) ; -- hbe_l(1:0)
    END IF ;
------------------------------------------------------------------------
  WHEN pt_hpi_s2 =>
------------------------------------------------------------------------
    IF (ptwr = false_h) THEN
      ao_wr_l <= true_l ;
    END IF ;
    hds1_l <= true_l ;
------------------------------------------------------------------------
  WHEN pt_hpi_s3a =>
------------------------------------------------------------------------
    NULL ;
------------------------------------------------------------------------
  WHEN pt_hpi_s3 =>
------------------------------------------------------------------------
    IF (hrdy = true_h) THEN
      ao_wr_l <= false_l ;
      ao_be_l <= x”3” ;
      hds1_l <= false_l ;
      tbca_hpic(2) <= true_h ;   -- (hhwil) second access
      IF (ptwr = true_h) THEN
        tbca_hpic(1 DOWNTO 0) <= pt_be_l(3 DOWNTO 2) ;  -- hbe_l(1:0)
      END IF ;
    END IF ;
------------------------------------------------------------------------
  WHEN pt_hpi_s4 =>
------------------------------------------------------------------------
    IF (ptwr = false_h) THEN
```

```
          ao_wr_l <= true_l ;
        END IF ;
        hds1_l <= true_l ;
    ------------------------------------------------------------------------
      WHEN pt_hpi_s5a =>
    ------------------------------------------------------------------------
        NULL ;
    ------------------------------------------------------------------------
      WHEN pt_hpi_s5 =>
    ------------------------------------------------------------------------
        IF (hrdy = true_h) THEN
          ao_wr_l <= false_l ;
          hds1_l <= false_l ;
          pt_rdy_l <= true_l ;
        END IF ;
    ------------------------------------------------------------------------
      WHEN pt_hpi_s6 =>
    ------------------------------------------------------------------------
        ao_be_l <= ”1111” ;
--        ao_adr <= ”00000” ; -- rev 3
        ao_sel_l <= false_l ;
        ao_rd_l <= false_l ;
        pt_rdy_l <= false_l ;
        hcs_l <= false_l ;
        tbca_hpic <= ”00011” ;
    ------------------------------------------------------------------------
      WHEN pt_hpi_s7 =>
    ------------------------------------------------------------------------
        IF ((ptnum = hpib_acc) AND      -- looping back to pt_hpi_s1
            (ptatn_l = true_l)) THEN
--          ao_adr <= pci_aptd_adr ; -- rev 3
          ao_sel_l <= true_l ;
          ao_be_l <= x”C” ;
          IF (ptwr = true_h) THEN
            ao_rd_l <= true_l ;
          END IF ;
        END IF ;
    ------------------------------------------------------------------------
      WHEN done =>
    ------------------------------------------------------------------------
        ao_bsy <= false_h ;
        ao_sel_l <= false_l ;
        ao_rd_l <= false_l ;
        rdfifo_l <= false_l ;
        ao_be_l <= ”0000” ;
--        ao_adr <= ”00000” ; -- rev 3
        adr_mux_sel <= true_h ;  -- rev 3
        pt_rdy_l <= false_l ;
        tbc_rd_l <= false_l ;
        pcireg_oe <= false_h ;
    ------------------------------------------------------------------------
      WHEN OTHERS =>
    ------------------------------------------------------------------------
        NULL ;
    END CASE ;
  END IF ; -- clocked IF
END PROCESS seq ;

END rtl ;
```

Appendix D

TMS320C62x McEVM PCI Configuration EEPROM

This appendix documents the EEPROM used on the ’C62x McEVM to initialize the AMCC S5933 peripheral component interconnect (PCI) controller’s configuration space registers.

The S5933 PCI controller provides the configuration registers required to support the PCI’s plug-and-play capability. The initialization of these registers is performed upon power up and system reset by the S5933, which clocks serial data in from the EEPROM using a two-wire, bidirectional data transfer. This automatic register initialization is performed since the S5933’s SNV pin is pulled up on the McEVM to indicate that a serial nonvolatile memory device is present.

Table D–1 summarizes the contents of the 1K-byte EEPROM. Multibyte default values are stored in least-significant-byte to most-significant-byte (little-endian) order. The 64-byte configuration information is stored in the EEPROM locations 0x40 to 0x7F. Locations 0x00 to 0x3F are reserved for future use by Texas Instruments. Locations 0x80 to 0x3FF are available for user-defined, nonvolatile storage that is accessible by both host and DSP software.

Table D–1. PCI Configuration EEPROM Summary

EEPROM Offset	Register Initialized	Description	Value	Selection
0x00–0x3F	–	Reserved for future use	All 0x00	–
0x40–0x41	VID	Vendor identification	0x104C	TI
0x42–0x043	DID	Device identification	0x1003	McEVM
0x44	–	Reserved	0x00	–
0x45	–	FIFO configuration	0xE1	PCI, async
0x46–0x47	–	Reserved	0x0000	–
0x48	RID	Revision identification register	0x00	Rev. 0 board
0x49–0x4B	CLCD	Class code register	0x0B4000	Coprocessor
0x4C	–	Reserved	0x00	–
0x4D	LAT	Master latency timer	0xF8	248 clocks
0x4E	HDR	Header type	0x00	One function
0x4F	BIST	Built-in self test	0x00	No BIST
0x50–0x53	BAR0	Base address register 0	0x10E8FFC0	16 DWORDs
0x54–0x57	BAR1	Base address register 1	0xFFFFFF80	32 DWORDs
0x58–0x5B	BAR2	Base address register 2	0xFFFFFF80	32 DWORDs
0x5C–0x5F	BAR3	Base address register 3	0xBFFFFFF0	4 DWORDs
0x60–0x63	BAR4	Base address register 4	0xBFFC0000	64K DWORDs
0x64–0x67	BAR5	Base address register 5	0x00000000	Not used
0x68–0x6F	–	Reserved	All 0x00	–
0x70–0x73	EXROM	Expansion ROM base address	0x00000000	No ROM
0x74–0x7B	–	Reserved	All 0x00	–
0x7C	INTLN	Interrupt line	0xFF	Autoassign IRQ
0x7D	INTPIN	Interrupt pin	0x01	INTA#
0x7E	MINGNT	Minimum grant	0x00	No min grant
0x7F	MAXLAT	Maximum latency	0x00	No max latency
0x80–0x3FF[†]	–	Not used (available)	All 0x00	–

[†] These address offsets are available for general-purpose use. This is a total of 896 bytes.

Appendix E

Glossary

A

A/D: See *analog-to-digital.*

adaptive differential pulse code modulation (ADPCM): A speech coding method that calculates the difference between two consecutive speech samples and encodes it using an adaptive filter to transmit at a lower rate than the standard 64-kbps pulse code modulation technique.

ADC: See *analog-to-digital converter*.

address: The logical location of program code or data stored in memory.

administrative privileges: Authority to set software and hardware access; includes access and privileges to install, manage, and maintain system and application software and directories on a network server or individual computer systems.

ADPCM: See *adaptive differential pulse code modulation*.

A-Law companding: See *compress and expand (compand)*.

ALU: See *arithmetic logic unit.*

American National Standards Institute (ANSI): A standards-setting, non-government organization that develops and publishes standards for voluntary use in the United States.

American Standard Code for Information Interchange (ASCII): A standard computer code for representing and exchanging alphanumeric information.

analog-to-digital (A/D): Conversion of continuously variable electrical signals to discrete or discontinuous electrical signals.

analog-to-digital converter (ADC): A converter with internal sample-and-hold circuitry used to translate an analog signal to a digital signal.

ANSI C: A version of the C programming language that conforms to the C standards defined by the American National Standards Institute.

application programming interface (API): A set of standard software function calls and data formats that application programs use to interact with other applications, device-specific drivers, or the operating system.

application-specific integrated circuit (ASIC): A custom chip designed for a specific application. It is designed by integrating standard cells from a library.

arithmetic logic unit (ALU): The section of the computer that carries out all arithmetic operations (addition, subtraction, multiplication, division, or comparison) and logic functions.

ASCII: See *American Standard Code for Information Interchange*.

ASIC: See *application-specific integrated circuit.*

assembler: A software program that creates a machine language program from a source file that contains assembly language instructions, directives, and macros. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assert: To make a digital logic device pin active. If the pin is active low, then a low voltage on the pin asserts it. If the pin is active high, then a high voltage asserts it.

B

ball grid array (BGA): An integrated circuit package in which the input and output connections are solder balls arranged in a grid pattern.

base address register (BAR): A device configuration register that defines the start address, length, and type of memory space required by a peripheral component interconnect (PCI) device. The value written to this register during device configuration programs its memory decoder to detect accesses within the indicated range.

basic input/output system (BIOS): A firmware program that is responsible for power-on testing and initialization of a computer. In addition, it may provide runtime services for operating systems.

BBS: See *bulletin board service*.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

BGA: See *ball grid array*.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian.*

BIOS: See *basic input/output system*.

bit: A binary digit, either 0 or 1.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The 'C62x DSP supports booting from external ROM or the host port interface (HPI).

bulletin board service (BBS): An electronic bulletin board that allows users to post and read messages and download software.

bus master: A device capable of initiating a data transfer with another device.

byte: A sequence of eight adjacent bits operated upon as a unit.

C

CBT: See *crossbar technology*.

CD-ROM: See *compact disc read-only memory*.

central processing unit (CPU): The CPU is the portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

channel service unit (CSU): A device used to connect a digital phone line, such as T1/E1, coming in from the phone company to another device producing a digital signal.

clock cycle: A cycle based on the input from the external clock.

clock mode (clock generator): One of the modes that sets the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal CLKIN.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

CMOS: See *complementary metal oxide semiconductor*.

coder-decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

common object file format (COFF): A system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space. The ’C62x code generation tools generate COFF files.

compact disc read-only memory (CD-ROM): A 4.7-inch optical disk that can hold as much as 660M bytes of digital data. A CD-ROM can store digitized audio, image, video, text, and application data.

compiler: A translation program that converts a high-level language set of instructions into a target machine’s assembly language.

complementary metal oxide semiconductor (CMOS): An integrated circuit technology that uses complementary transistors to efficiently charge and discharge capacitive loads in both the positive and negative directions and dissipates power only on transitions.

complex programmable logic device (CPLD): A digital, user-configurable integrated circuit used to implement custom logic functions.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and then, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes—A-law, used in Europe, and μ-law, used in the United States.

CPLD: See *complex programmable logic device*.

crossbar technology (CBT): High-speed bus-connect devices that are useful for bus isolation, multiplexing, and voltage translation. These devices have an on-state resistance of 5 ohms and a propagation delay of 250 ps.

CPU: See *central processing unit*.

CSU: See *channel service unit*.

D

D/A: See *digital-to-analog.*

DAC: See *digital-to-analog converter*.

daughterboard: A circuit board that connects to a motherboard to provide additional capabilities and/or interfaces. See also *motherboard*.

dB: See *decibels*.

debugger: A software interface used to identify and eliminate mistakes in a program.

decibels (dB): A unit for measuring the level of signal relative to a defined reference signal that follows it. For example, the notation dBm indicates a signal power level relative to a 1 milliwatt reference signal.

device driver: Software that enables computer hardware to communicate with a device. A device driver may also translate data and call other drivers to actually send data to a device.

device ID: Every peripheral component interconnect (PCI) device must have a device ID configuration register to identify itself.

digital signal processor (DSP): A semiconductor that turns analog signals—such as sound or light—into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

digital-to-analog (D/A): Conversion of discrete or discontinuous electrical signals to continuously variable signals. See also *digital-to-analog converter*.

digital-to-analog converter (DAC): A device that converts a signal represented by a series of numbers (digital) to a continuously varying signal (analog). See also *digital-to-analog*.

DIP: See *dual in-line package*.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for, and receives, mastery of the memory bus so that data transfers can take place independent of the host.

DLL: See *dynamic link library*.

DMA: See *direct memory access*.

doubleword (DWORD): The PCI (host) defines a doubleword as a 32-bit value. See also *word* and *halfword*.

driver: See *device driver*.

DSP: See *digital signal processor.*

dual in-line package (DIP): A common rectangular chip housing with leads (pins) on both long sides.

DWORD: See *doubleword*.

dynamic link library (DLL): A Windows software library that is linked dynamically at run time, rather than statically at compile time. DLLs can be shared among multiple applications and be replaced with newer versions without requiring the applications to be recompiled.

E

EEPROM: See *electrically-erasable programmable read-only memory*.

electret microphone: A condenser microphone that requires an external power source.

electrically-erasable programmable read-only memory (EEPROM): A nonvolatile memory device that can be programmed in-circuit and have its contents selectively changed. Although its name includes *read-only*, it supports both read and write accesses.

electrostatic discharge (ESD): Discharge of a static charge on a surface or body through a conductive path to ground, which can be damaging to integrated circuits.

EMIF: See *external memory interface*.

erasable programmable read-only memory (EPROM): A nonvolatile memory device that can be erased with exposure to ultraviolet light. The device can be randomly accessed, but it is read only.

ESD: See *electrostatic discharge*.

evaluation module (EVM): A board and software tools that allow the user to evaluate a specific device.

expansion interface: An interface that allows additional capabilities to be added to a base product.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

external memory interface (EMIF): The boundary between the CPU and external memory through which information is conveyed.

F

first in, first out (FIFO): A queue; a data structure or hardware buffer from which items are taken out in the same order they were put in. A FIFO is useful for buffering a stream of data between a sender and receiver that are not synchronized; that is, the sender and receiver are not sending and receiving at exactly the same rate. If the rates differ by too much in one direction for too long, the FIFO becomes either full (blocking the sender) or empty (blocking the receiver).

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

Flash memory: Nonvolatile read-only memory that is electronically erasable and programmable.

flexible MVIP integrated circuit (FMIC): Device which provides a complete MVIP-compliant interface between the MVIP bus and a variety of processors, telephony interfaces and other circuits. It also provides a 384 x 384 switching matrix between the MVIP bus and four local data streams.

H

halfword: The ’C62x DSP defines a halfword as a16-bit data value. See also *doubleword* and *word*.

handle: An identifier used by software to reference a file or device.

high-level language (HLL): A general-purpose language that can be used to program a microprocessor rather than using a low-level, machine-dependent language.

host: A device to which other devices (peripherals) are connected and that generally controls those devices.

host port interface (HPI): A 16-bit parallel interface that the host uses to access the DSP’s memory space.

I

identifier (ID): A field that contains a resource-table index, a sequence number, and a resource-type code; it identifies a kernel resource such as a port, semaphore, or task.

IEEE 1149.1 standard: “IEEE Standard Test Access Port and Boundary-Scan Architecture”, first released in 1990. See also *JTAG*.

Industry Standard Architecture (ISA): An industry-standard 8/16-bit bus used in IBM™ compatible desktops. It provides a theoretical maximum data transfer rate of 8.33M bytes per second.

initiator: When a PCI bus master has arbitrated for and won access to the PCI bus, it becomes the initiator of a transaction.

Institute of Electrical and Electronic Engineers (IEEE): A publishing and standards-making body focused on advancing the theory and practice of electrical, electronics, computer engineering, and computer science.

in-system programmable (ISP): The ability to program and reprogram a device on a circuit board.

Integrated Services Digital Network (ISDN): A worldwide digital communications network evolving from existing telephone services. Its goal is to replace current telephone lines that require DA conversions with totally digital switching and transmission facilities capable of carrying a variety of data—from voice to computer transmissions, music, and video. The ISDN is built on two main types of communications channels: a B channel, which carries data at 64 Kb/s, and a D channel, which carries control information at either 16 or 64 Kb/s. Computers and other devices connect ISDN lines through simple, standardized interfaces.

integrated switching regulator (ISR): A complete switch-mode power supply in a modular, board-mounted package.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service routine: A module of code that is executed in response to a hardware or software interrupt.

ISDN: See *Integrated Services Digital Network*.

ISP: See *in-system programmable*.

ISR: See *integrated switching regulator*.

J

Joint Test Action Group (JTAG): The Joint Test Action Group was formed in 1985 to develop economical test methodologies for systems designed around complex integrated circuits and assembled with surface-mount technologies. The group drafted a standard that was subsequently adopted by IEEE as IEEE Standard 1149.1-1990, "IEEE Standard Test Access Port and Boundary-Scan Architecture."

K

kilohertz (kHz): One thousand hertz, or cycles per second, used to indicate the frequency of a clock signal.

L

latch phase: The phase of a CPU cycle during which internal values are held constant.

LBO: See *line build out*.

light emitting diode (LED): A semiconductor chip that gives off visible or infrared light when activated.

line build out (LBO): Selectable output attenuation with typical loss of 0.0, 7.5 and 15 dB at 772 kHz.

line interface unit (LIU): A device or a part of a device responsible for interfacing to a digital network (T1/E1) line.

linker: A software tool that combines object files to a form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

LIU: See *line interface unit*.

load: To enter data into storage or working registers.

loader: A device that places an executable module into system memory.

M

mA: See *milliamp.*

macro: A sequence of statements or instructions that is represented by a symbolic symbol.

μ**F:** See *microfarad*.

μ**-Law companding:** See *compress and expand (compand)*.

mailbox: A 32-bit register that provides a simple communication method to pass messages between the host and DSP software. Multiple mailboxes are typically available to support bidirectional, multiword message transfers.

maskable interrupt: An interrupt that can be enabled or disabled through software.

master clock output signal (CLKOUT1): The output signal of the on-chip clock generator. The CLKOUT1 high pulse signifies the CPU's logic phrase (when internal values are changed), while the CLKOUT1 low pulse signifies the CPU's latch phase (when the values are held constant).

Mbps: See *megabit per second*.

McBSP: See *multichannel buffered serial port.*

McEVM: A multichannel communications development board and software tools that allow the user to evaluate a specific device.

megabit per second (Mbps): A million bits of data per second.

megahertz (MHz): One million hertz, or cycles per second, used to indicate the frequency of a clock signal.

memory map: A graphical representation of a computer system's memory, showing the locations of program space, data space, reserved space, and other memory-resident elements.

memory-mapped register: An on-chip register mapped to an address in memory. Some memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

MHz: See *megahertz*.

microfarad (μ**F):** One-millionth of a farad, which is the basic unit of capacitance.

microsecond (µs): One-millionth of a second.

milliamp (mA): One-thousandth of an ampere, which is the basic unit of current.

millimeter (mm): One-thousandth of a meter.

million instructions per second (MIPS): A unit of instruction execution speed of a computer.

millisecond (ms): One-thousandth of a second.

millivolts root mean square (mV$_{rms}$): One-thousandth of a volt root mean square. See also *volts root mean square.*.

MIPS: See *million instructions per second*.

mm: See *millimeter*.

most significant byte (MSbyte): The byte in a multibyte word that has the most influence on the value of a word.

motherboard: The main circuit board that contains the processor, main memory, circuitry, bus controller, connectors, and primary components of the computer. See also *daughterboard*.

µ**s:** See *microsecond*.

ms: See *millisecond*.

MSbyte: See *most significant byte*.

multichannel buffered serial port (McBSP): A standard serial port interface found on ’C62x devices. It provides full-duplex communication, double-buffered data registers, independent transmit and receive framing and clocking, direct interface to industry-standard serial devices, internal and external clock support, and an autobuffering capability using a DMA controller.

multiplexing: A process of transmitting more than one set of signals at a time over a single wire or communications link. (Also known as muxing.)

Multi-Vendor Integration Protocol (MVIP): A family of standards that allows products from different vendors to interoperate withina computer or group of computers. The MVIP bus is a telephony bus that provides 256 full-duplex voice channels (16 streams of 32 timeslots) over a ribbon cables between PC boards.

mutex: A mutual exclusion semaphore used to restrict access to a resource.

MVIP: See *Multi-Vendor Integration Protocol*.

mV$_{rms}$: See *millivolts root mean square*.

N

nanosecond (ns): One-billionth of a second, the basic unit of time.

nonmaskable interrupt (NMI): An interrupt that uses the same logic as the maskable interrupts, but can be neither masked nor disabled. It is often used as a soft reset.

nonvolatile random access memory (NVRAM): A type of random access memory that retains its data when its power source is turned off, providing nonvolatile storage.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A device external to the device.

on chip: An element or module internal to the device.

P

parallel debug manager (PDM): A program used for creating and controlling multiple debuggers for the purpose of debugging code in a parallel-processing environment.

PC: *Personal computer*.

PCI: See *peripheral component interconnect*.

PCM: See *pulse code modulation*.

PDM: See *parallel debug manager*.

peripheral component interconnect (PCI): A high-speed local bus that supports data-transfer speeds of up to 132M bytes per second at 33 MHz.

phase-locked loop (PLL): A unit within a system that uses phase to lock on to a signal to ensure synchronous clocking of digital signals.

pitch: The distance between successive centers of leads of a component package.

plastic quad flat pack (PQFP): A low-profile, surface-mount integrated circuit package that is plastic and has leads (pins) on all four sides.

PLL: See *phase-locked loop*.

poll: A continuous test used by the program until a desired condition is met.

PQFP: See *plastic quad flat pack.*

profiling environment: A special debugger environment that provides a method for collecting execution statistics about specific areas in application code.

pulse code modulation (PCM): The most common method of encoding an analog voice signal into digital data. Voice signals are encoded into 8-bit data samples at an 8-kHz sample rate, resulting in a 64-kbps digital data stream.

R

random-access memory (RAM): A memory element that can be written to as well as read from.

read-only memory (ROM): A semiconductor storage element containing permanent data that cannot be changed.

ready: A task state indicating that the task either is currently executing or is able to execute as soon as it acquires the processor.

real time: The actual time during which the physical process of computation transpires in order that results of the computation interact with a physical process.

reduced instruction set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

register: A small area of high-speed memory, located within a processor or electronic device, that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reset: A means to bring processors to known states by setting registers and control bits to predetermined values and signaling execution to start at a specified address.

ring 0: Highest level of privilege available on an Intel processor that defines what data can be accessed, what code in memory can be executed, and what machine instructions can be executed by a program. Low-level device drivers run at ring 0. See also *ring 3.*

ring 3: Lowest level of privilege available on an Intel processor that defines what data can be accessed, what code in memory can be executed, and what machine instructions can be executed by a program. High-level user-mode applications and DLL run at ring 3. See also *ring 0.*

RISC: See *reduced instruction set computer*.

ROM: See *read-only memory*.

S

sample rate: The rate at which the audio codec samples audio data. Usually specified in hertz (samples per second).

SBSRAM: See *synchronous burst static random-access memor*y.

SDRAM: See *synchronous dynamic random-access memory.*

slave: Another name for the target being addressed during a PCI transaction.

static random-access memory (SRAM): Fast memory that does not require refreshing, as DRAM does. It is more expensive than DRAM, though, and is not available in as high a density as DRAM.

structure: A collection of one or more variables grouped together under a single name.

surface-mount technology: A method of assembling printed wiring boards where components are mounted onto the surface rather than through holes.

synchronous burst static random-access memory (SBSRAM): High-performance SRAM device with accesses that are synchronized to a microprocessor clock and includes a burst address counter.

synchronous dynamic random-access memory (SDRAM): High-performance DRAM device with accesses that are synchronized to a microprocessor clock and support for page bursts.

syntax: The grammatical and structural rules of a language. All higher-level programming languages possess a formal syntax.

T

T1/E1: T1 is a digital transmission link with a capacity of 1.544 Mbps. It uses two pairs of normal twisted-wires and can handle 24-voice conversations, each digitized using mu-law coding at 64 kbps. T1 is used in USA, Canada, Hong Kong, and Japan. E1 is a digital transmission link with a capacity of 2.048 Mbps. It is the European equivalent of T1. It can handle 30-voice conversations, each digitized using A-law coding at 64 kbps.

target : When related to PCI, it is the device that is the target of a PCI transaction initiated by a PCI bus master. When related to the debugger, the DSP is the target of an emulation access.

target memory: Physical memory in a device into which executable object code is loaded.

TDM: See *time-division multiplexed*.

test bus controller (TBC): Application-specific integrated circuit that controls an IEEE 1149.1–1990 (JTAG) serial-test bus to support production testing and in-system microprocessor emulation. The TBC provides control of the DSP and access to all of its registers and memory.

thin quad flat pack (TQFP): A very low-profile, surface-mount integrated circuit package that is plastic and has leads (pins) on all four sides.

thread of execution: A schedulable unit of execution in a multitasking system. The term refers specifically to the progressive execution of a program element; it excludes other attributes, such as the system resources allocated to a task or process.

time-division multiplexed (TDM): The process by which a single serial bus is shared by multiple devices with each device taking turns to communicate on the bus. The total number of time slots (channels) depends on the number of devices connected. During a time slot, a given device may talk to any combination of devices on the bus.

timer: A programmable peripheral used to generate pulses or to time events.

TQFP: See *thin quad flat pack.*

transistor-transistor logic (TTL): A family of logic devices that are made with bipolar junction transistors and resistors. A TTL low level is defined as a voltage level below 0.4 volts. A TTL high level is defined as a voltage level above 2.4 volts.

tri-state: High impedance.

V

V: See *volt.*

VBAP: See *voice-band audio processor*.

V_{dc}: See *volts direct current.*

VelociTI: Architecture developed by Texas Instruments that features very long instruction words.

vendor ID: Every PCI device must have a vendor ID configuration register that identifies the vendor of the device.

very long instruction word (VLIW): Architecture using words between the sizes of 256 bits and 1024 bits.

virtual device driver (VxD): A 32-bit, ring-0 module that virtualizes hardware for ring-3 modules to provide a primary interface to hardware or specialized software services.

VLIW: See *very long instruction word*.

voice-band audio processor (VBAP): A voice codec device that provides filtering, analog-to-digital and digital-to-analog conversion with interfaces to a microphone, speaker, and serial, digital itnerface. It can operate in either mu-law or A-law companding or 13-bit linear modes.

volt (V): The unit of voltage, or potential difference.

volts direct current (V_{dc}): The voltage measurement of a direct current signal.

volts peak-to-peak (V_{pp}): A measurement of a signal that indicates the difference between its maximum and minimum voltage values.

volts root mean square (V_{rms}): A measurement of a periodic signal that indicates the effective voltage at which a direct current voltage would deliver the same average power. This measurement provides a method for comparing the power delivered by different waveforms.

VxD: See *virtual device driver*.

W

Win32: The 32-bit application programming interface for both Windows 95 and Windows NT.

word: A character or bit string considered as an entity. The length of the word is machine-dependent. The ’C62x DSP defines a word as a 16-bit data value. The PCI (host) defines a word as a 16-bit data value. See also *doubleword* and *halfword*.

X

XDS510: A hardware emulator that provides a scan-path connection to a DSP for source code debugging.

xDSL: A generic name (x is the generic) for digital subscriber line equipment and services. Asymmetric DSL (ADSL) is one of the more popular types, providing up to 6 Mbsp downstream and 640 kbps upstream data rates using standard twisted pair wiring.

Index